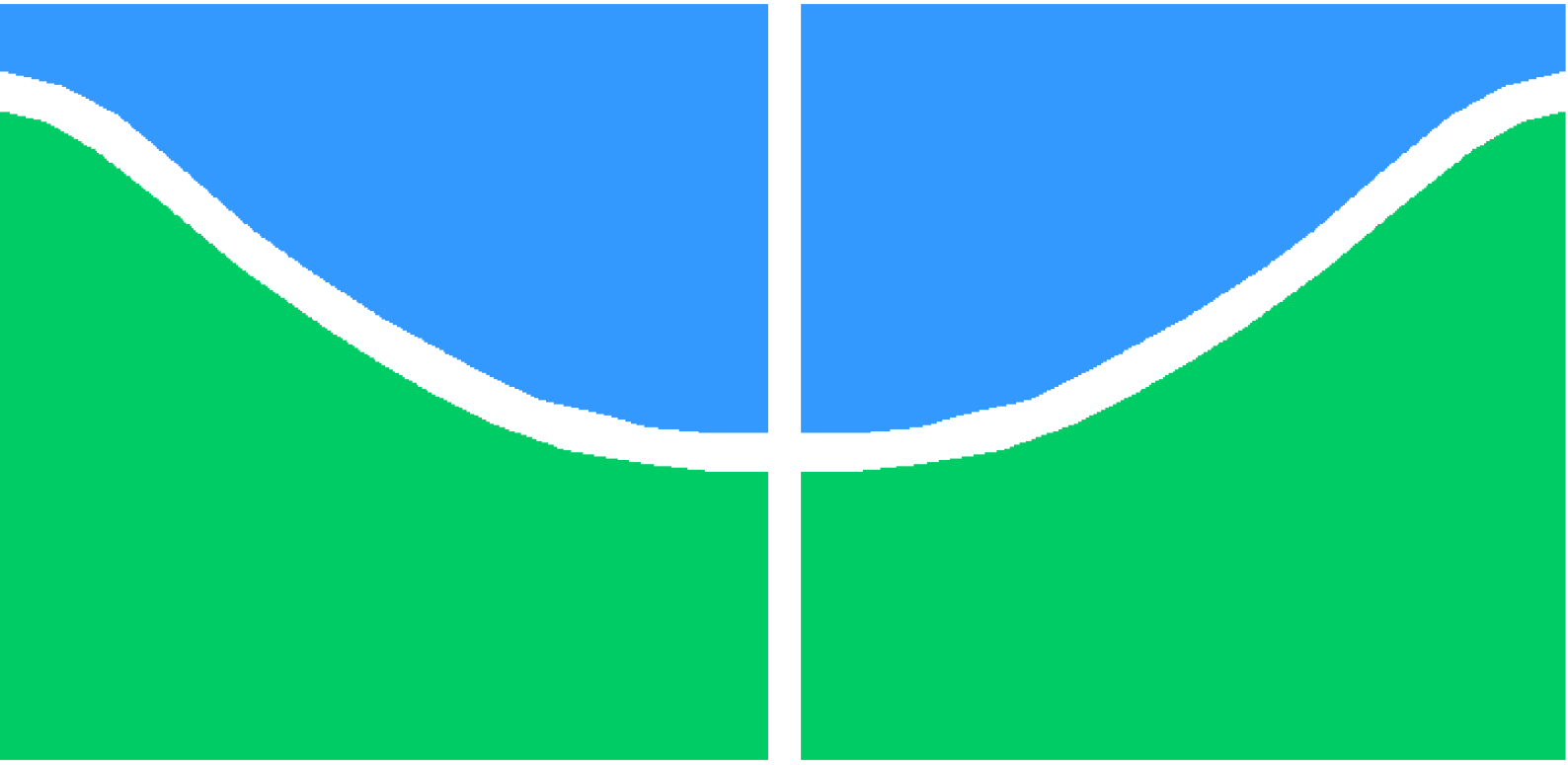

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia Eletrônica

PROJETO DE CONTROLADORES EMBARCADOS PARA UM QUADRIROTOR

Autor: Leonardo Avelino de Lima Jacinto
Orientador: Dr. Evandro Leonardo Silva Teixeira

Brasília, DF
2017

Leonardo Avelino de Lima Jacinto

PROJETO DE CONTROLADORES EMBARCADOS PARA UM QUADRIROTOR

Monografia submetida ao curso de graduação em (Engenharia Eletrônica) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia Eletrônica).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Dr. Evandro Leonardo Silva Teixeira

Brasília, DF

2017

Leonardo Avelino de Lima Jacinto
PROJETO DE CONTROLADORES EMBARCADOS PARA UM QUADRIROTOR/ Leonardo Avelino de Lima Jacinto. – Brasília, DF, 2017-
128 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Evandro Leonardo Silva Teixeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2017.

1. quadrirotor. 2. sistemas embarcados. I. Dr. Evandro Leonardo Silva Teixeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. PROJETO DE CONTROLADORES EMBARCADOS PARA UM QUADRIROTOR

CDU 02:141:005.6

Leonardo Avelino de Lima Jacinto

PROJETO DE CONTROLADORES EMBARCADOS PARA UM QUADRIROTOR

Monografia submetida ao curso de graduação em (Engenharia Eletrônica) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia Eletrônica).

Trabalho aprovado. Brasília, DF, 04 de Julho de 2017:

Dr. Evandro Leonardo Silva Teixeira
Orientador

Dr. Renato Vilela Lopes
Convidado 1

Dr. André Murilo de Almeida Pinto
Convidado 2

Brasília, DF
2017

Este trabalho é dedicado à minha mãe, Joselene,
ao meu pai, Almir, aos meus irmãos Lucas
e Luís, e à minha namorada Amanda.

Agradecimentos

Agradeço à minha família, meus pais e irmãos, por sempre acreditarem em mim e por todo investimento, carinho e suporte ao longo de todas as etapas da minha vida. Agradeço à minha namorada Amanda, que apesar da distância está sempre me apoiando, respeitando minhas decisões e me incentivando a sempre optar pelas melhores escolhas.

Agradeço ao meu tio Marcos, por todos os conselhos desde a infância até a fase adulta, por todo o apoio que precisei para tomar decisões importantes ao longo das etapas vividas.

Agradeço aos meus colegas de curso Rafael, Hugo e Paulo, pela amizade e companheirismo durante a minha jornada universitária. Agradeço ao meu colega Tiago que diversas vezes me ajudou a enxergar as melhores soluções que possibilitaram a execução deste projeto.

Agradeço ao professor Evandro pela oportunidade de trabalhar neste projeto e por todo o suporte fornecido por ele ao longo do processo de aprendizado. Agradeço aos professores André e Renato por disponibilizarem todo o material e ambiente do projeto e por estarem prontos à ajudar quando necessário.

"Success is not the key to happiness.
Hapiness is the key to success.
If you love what you are doing,
you will be successful."
(Albert Schweitzer)

Resumo

O presente trabalho aborda a implementação de controladores embarcados em um Veículo Aéreo Não-Tripulado (*VANT*) do tipo quadrirotor. Dois controladores bem conhecidos na área de sistemas de controle foram escolhidos: o controlador proporcional-integral-derivativo (*PID*) e o regulador linear quadrático (*LQR*). O objetivo é implementar estes controladores em uma famosa plataforma de controle de vôo conhecida como *PX4 Autopilot*, garantindo que mesmo com todas as aplicações já existentes, o controlador funcione de forma determinística cumprindo as restrições temporais. Um estudo sobre esta plataforma será apresentado, e a resposta dos controladores implementados serão comparadas com a resposta do controlador oficial da *PX4*. Por fim, serão apresentados os resultados alcançados em simulação e na prática.

Palavras-chaves: quadrirotor. sistemas embarcados. *PX4*.

Abstract

The present work addresses the implementation embedded controller in an Unmanned Aerial Vehicle (UAV) quadrotor type. Two well-known controller in control systems field were chosen: the proportional-integral-derivative (PID) controller and the linear quadratic regulator (LQR). The goal is to implement these two controllers in a famous flight control platform known as PX4 Autopilot, ensuring that, even with all the existing applications, the controller works in a deterministic manner complying with the time constraints. An study about this platform will be presented, and the results of the implemented controllers will be compared against the PX4's official controller. Lastly, the results achieved in both simulation and practice will be presented in order to demonstrate the success of the implemented control techniques embedded in a quadrotor.

Key-words: quadrotor. embedded systems. PX4.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

A/D	Analógico/Digital
API	*Application Programming Interface*
APM	*Ardupilot Meta*
ARM	*Advanced RISC Machine*
BSD	*Berkeley Software Distribution*
CAN	*Controller Area Network*
CPU	*Central Processing Unit*
DC	*Direct Current*
E/S	Entrada/Saída
ESC	*Electronic Speed Controller*
I2C	*Inter-Integrated Circuit*
IMU	*Inertial Measure Unit*
I/O	*Input/Output*
GPIO	*General Purpose Input/Output*
GPS	*Global Positioning System*
HIL	*Hardware-in-the-loop*
JTAG	*Joint Test Action Group*
LED	*Light Emitting Diode*
LIDAR	*Light Detection And Ranging*
LiPo	*Lithium-Polymer*
LQR	*Linear Quadratic Regulator*
MCU	Microcontrolador
MCU	*Model Predictive Control*

ORB	*Object Request Broker*
P	Proporcional
PC	*Personal Computer*
PD	Proporcional Derivativo
PID	Proporcional Integral Derivativo
POSIX	*Portable Operating System Interface*
PWM	*Pulse Width Modulation*
RAM	*Random Access Memory*
RC	Radio Controle (*Radio Controller*)
RF	Radio Frequência (*Radio Frequency*)
RPM	Rotações Por Minuto
RTOS	*Real Time Operating System*
SD	*Secure Digital*
SIL	*Software-in-the-loop*
SPI	*Secure Peripheral Interface*
UART	*Universal Asynchronous Receiver Transmitter*
UAV	*Unmanned Aerial Vehicle*
UnB	Universidade de Brasília
uORB	*Micro Object Request Broker*
USB	*Universal Serial Bus*
TCC	Trabalho de Conclusão de Curso
UAV	Unmanned Aerial Vehicle
VANT	Véiculo Aéreo Não-Tripulado

Lista de símbolos

A	Matriz de estados
b	Coeficiente de arrasto
B	Matriz de entradas
C_x	Cosseno de x
$\boldsymbol{f}$	Vetor com a força aplicada ao quadrirotor devido ao empuxo
f_i	Força na direção do eixo do motor i
g	Constante gravitacional
$\boldsymbol{I}$	Matriz dos momentos de inércia
I_{xx}	Momento de inércia em torno do eixo x
I_{yy}	Momento de inércia em torno do eixo y
I_{zz}	Momento de inércia em torno do eixo z
$\boldsymbol{J}$	Matriz jacobiana de rotação expressa em termos das coordenadas generalizadas
J	Função custo
$\boldsymbol{K}$	Matriz constante de ganhos
k	Coeficiente de empuxo
l	Distância entre o eixo do motor e centro de massa do corpo
m	Massa do corpo
P	Matriz para cálculo da matriz de ganho K
p	Velocidade angular no eixo x
$\boldsymbol{Q}$	Matriz de penalidade
$\boldsymbol{q}$	Vetor com as coordenadas generalizadas no referencial inercial
q	Velocidade angular no eixo y
r	Velocidade angular no eixo z

$\boldsymbol{R}$	Matriz de Penalidade
S_x	Seno de x
T	Soma de todos os empuxos exercidos por cada um dos motores (*throttle*)
$u(k)$	Vetor de entradas discretizado
$\boldsymbol{u}(\mathrm{t})$	Vetor de entradas
$\mathbf{V}_\mathrm{B}$	Velocidades lineares no referencial do corpo
w_i	Velocidade angular do motor i
x	Eixo de translação ou coordenada do quadrirotor ao longo deste eixo
$\boldsymbol{x}(\mathrm{t})$	Vetor de estados
y	Eixo de translação ou coordenada do quadrirotor ao longo deste eixo
z	Eixo de translação ou coordenada do quadrirotor ao longo deste eixo
$\boldsymbol{\eta}$	Posições angulares no referencial inercial
θ	Ângulo de arfagem (*pitch*)
τ_{M_i}	Torque gerado ao redor do motor i
$\boldsymbol{\tau_b}$	Vetor de torques em cada eixo do referencial do corpo
$\boldsymbol{\nu}$	Velocidades angulares no referencial do corpo
$\boldsymbol{\xi}$	Posições lineares no referencial inercial
$\sigma(k)$	Vetor de estados discretizado
$\boldsymbol{\tau}$	Vetor com os torques no eixo x,y,z (torques em *roll*, *pitch*, *yaw*)
τ_s	Período de amostragem
τ_θ	Torque no eixo y
τ_ϕ	Torque no eixo x
τ_ψ	Torque no eixo z
ϕ	Ângulo de rolamento (*roll*)
ψ	Ângulo de guinada (*yaw*)
$\mathcal{L}$	Lagrangiano

Sumário

1 Introdução

Devido ao avanço da tecnologia na área de robótica e aviação, o uso de Veículos Aéreos Não-Tripulados (*VANTs*), também chamado de *drones*, para diversos propósitos é uma realidade do século XXI. Mesmo que, no início de seu desenvolvimento, os *VANTs* tenham sido usados para propósitos militares, atualmente a demanda é alta para uso cívil e comercial, incluindo setores como agricultura, construção, inspeção e segurança [53]. De acordo com o jornal *O Globo*, o mercado de *VANTs* é estimado em 127 bilhões de dólares ao ano em todo o mundo[41].

Além de proporcionar uma grande movimentação financeira, o avanço da tecnologia faz com que os *VANTs* sejam cada vez mais úteis para a humanidade. Em [44] pode-se ver *drones* gerando redução de gastos de insumos na agricultura. Em [26] é apresentado uma pesquisa em que é mostrado como *drones* podem salvar vidas ao carregar desfibriladores cardíacos. Em [19] foi registrado o primeiro vôo para a entrega de um pacote pela *Amazon*, maior empresa varejista de *e-commerce* do mundo.

Os *VANTs* só chegaram a este ponto em termos de tecnologia graças aos avanços da eletrônica. *Chips* microcontroladores e baterias de lítio tornaram-se mais baratos e menores recentemente, tornando-se passíveis de serem instalados nestes veículos [53]. Quadrirotores, categoria de *VANTs* de quatro motores cujas velocidades são controladas computacionalmente (exemplo na Figura 1), surgiram no começo dos anos 2000, e trouxeram uma nova maneira de realizar vôos mais ágeis e em espaços limitados, graças as técnicas de sistemas de controle desenvolvidas e aplicadas à este sistema por pesquisadores.

A estabilização e vôo de um *VANT* quadrirotor é puramente dependente de *software* que implementa a estratégia de controle desse sistema e é executado em um mi-

Figura 1 – Exemplo de *VANT* do tipo quadrirotor

crocontrolador embarcado no veículo. O microcontrolador deve ser capaz de executar múltiplos algoritmos para estimar dados dos sensores, executar cálculos, enviar comando aos motores, além de diversas outras aplicações, de forma a garantir o cumprimento das restrições de tempo, caso contrário o veículo pode cair. Tamanha complexidade é díficil de ser implementado com programação *bare metal*[1], o que é comumente adotado em sistemas embarcados com microcontroladores.

Conforme discutido em [31], sistemas robóticos que possuem aplicações complexas como os quadrirotores e demais *VANTs* devem utilizar um sistema operacional de tempo-real (*RTOS*, em inglês) proporcionando um ambiente seguro e confiável de desenvolvimento. Com o uso de um *RTOS* é possível gerenciar à execução das aplicações de forma a obedecer as restrições de tempo de controle, uma vez que se conhece bem o sistema embarcado em questão (tanto o *hardware* quanto o *software*). Com as devidas técnicas é possível embarcar algoritmos complexos de controle e demais aplicações de maneira otimizada, e garantir que o sistema funcione corretamente, como pode ser visto em diversas plataformas para controle de *VANTs* como *PX4 Autopilot* e *ArduPilot Mega* (*APM*).

A *PX4 Autopilot* é um projeto *open source* de *hardware* aberto bastante popular entre pesquisadores de *VANTs* [48]. Seu *hardware* mais popular, a *Pixhawk*, é uma controladora de vôo que possui um microcontrolador embarcado executando um *RTOS* para gerenciar suas aplicações. Por ter uma arquitetura funcional sendo desenvolvido pela comunidade internacional, o desenvolvimento para esta plataforma é interessante pois permite a reutilização de módulos [39]. Esta característica permite que o pesquisador tenha foco em seu projeto, sem se preocupar em desenvolver outras aplicações fora do escopo do mesmo. A *PX4* oferece ainda, ambientes de simulação que torna o desenvolvimento de aplicações de controle menos suscetível à falha em campo.

1.1 Objetivo

O objetivo deste trabalho é implementar duas estratégias de controle em um microcontrolador que utiliza um sistema operacional de tempo-real (*RTOS*), embarcado em um veículo do tipo quadrirotor de maneira otimizada computacionalmente, garantindo que as restrições de tempo sejam cumpridas. A plataforma *PX4 Autopilot*, que inclui o *firmware PX4* e o *hardware* (placa de controle de vôo) *Pixhawk*, será utilizada. Deseja-se utilizar diferentes estratégias de controle que serão discutidas no decorrer do trabalho, à fim de comparar o custo computacional de cada uma, assim como levantar as diferenças entre elas e as aplicações de controle já existentes desenvolvidas em plataformas utilizadas atualmente.

[1] Programação *bare metal* é um termo usado quando a aplicação interage diretamente com o *hardware*, sem um sistema operacional para intermediar

1.2 Estrutura do Trabalho

O texto deste trabalho foi organizado em seis capítulos, referências bibliográficas e apêndice. A seguir é apresentado um breve resumo do que o leitor pode deverá encontrar em cada um destes capítulos:

Capítulo 2: Neste capítulo é abordada a revisão bibliográfica que serviu de base para a execução deste trabalho. Uma introdução aos quadrirotores, além de uma revisão de cada uma das principais partes que compõe este veículo foi elaborada neste capítulo. Técnicas de controle destes, além de modelos já existentes e uma revisão de sistemas de tempo-real são também abordadas.

Capítulo 3: Neste capítulo é apresentada uma revisão sobre a plataforma *PX4 Autopilot*. Um estudo sobre os módulos de *software*, a controladora de vôo *Pixhawk* e simuladores é retratado.

Capítulo 4: Neste capítulo é abordada a metodologia para a execução deste trabalho. É descrito passo a passo os métodos que foram utilizados durante o desenvolvimento deste projeto.

Capítulo 5: Este capítulo trata da descrição dos estudos de caso deste trabalho. O primeiro estudo de caso trata da implementação de um controlador *PID* em um quadrirotor. Já o segundo, trata-se da implementação de um controlador LQR para o quadrirotor. Por fim é apresentado os resultados experimentais assim como a análise dos mesmos.

Capítulo 6: Este capítulo apresenta a conclusão do projeto.

2 Revisão Bibliográfica

Esta seção apresenta importantes conceitos de Veículos Aéreos Não-Tripulados do tipo quadrirotor, incluindo discussões sobre o modelo matemático, controle e os componentes do seu sistema computacional embarcado, que serviram de base para a elaboração deste trabalho.

2.1 Quadrirotor

2.1.1 Introdução

O quadrirotor, também conhecido como quadrotor ou quadricóptero, é um tipo de helicoptéro que possui quatro rotores, cada um com uma hélice. Embora seu uso tenha começado como aeronaves para passageiros no começo do século XX [50], hoje em dia, o mesmo é usado largamente como Veículo Aéreo Não-Tripulado. Estes veículos estão cada vez mais populares, principalmente devido as avançadas técnicas de controle desenvolvidas e o avanço da robótica, o que possibilita que eles sejam controlados por computadores em terra, por controles de rádio, ou até mesmo fazer vôos autônomos sem a necessidade de um piloto.

Várias são as aplicações dos quadrirotores nos dias de hoje como: na área militar para a procura de bombas e identificação de alvos, na área de segurança com o monitoramento de propriedades privadas, rastreamento de veículos nas estradas e também monitoramento de cargas. O uso também é bastante frequente para análise da atmosfera para previsão do tempo, detecção de áreas de incêndio e até mesmo para transporte de objetos [32].

Uma das vantagens do quadrirotor é que este é um veículo simétrico, sendo de simples construção, e possui uma certa facilidade de manutenção. Possui também algumas vantagens que o *VANT* de asa fixa não possui como decolagem vertical, maior facilidade de manobra, vôos em baixa velocidade, e também, consegue executar o movimento de pairar sob o ar (*hovering*). Além disso, o quadrirotor não precisa de uma área grande para aterrisagem, o que é necessário no caso do *drone* de asa fixa.

As lâminas das hélices do quadrirotor são conectadas nos rotores paralelamente entre si, fixando-as de maneira a obter uma estrutura rígida, permitindo que o único parâmetro a ser variado seja a velocidade de rotação das mesmas [21]. Ao variar a velocidade, os rotores geram a propulsão necessária para o vôo.

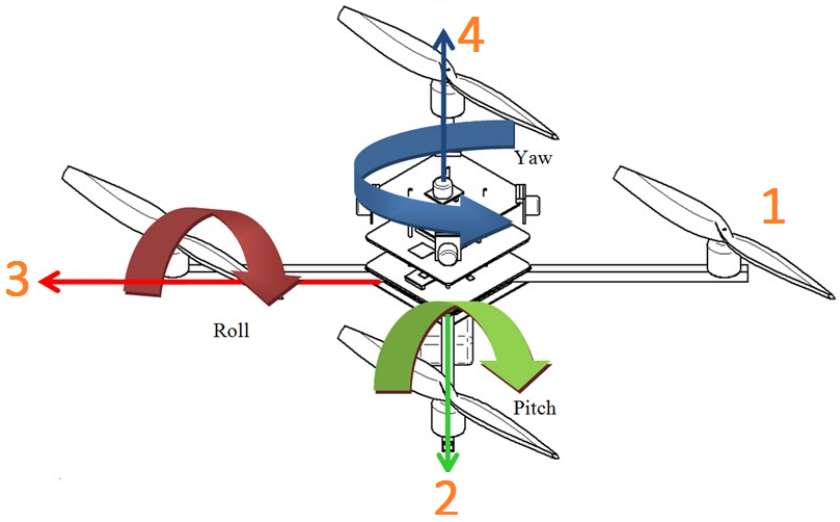

Figura 2 – Quadrirotor e seus movimentos de *roll*, *pitch* e *yaw*. Imagem adaptada de [1].

2.1.2 Princípio de Funcionamento de um Quadrirotor

A mecânica do vôo de um quadrirotor é possível graças aos diferentes sentidos de rotação de seus rotores como pode ser visto na Figura 2. Neste exemplo de configuração, a rotação das hélices 1 e 3 movem-se no sentido anti-horário, ao passo que as hélices 2 e 4 são rotacionadas no sentido horário. Desta forma, quando giram na mesma velocidade, os motores produzem um torque resultante nulo permitindo que o veículo fique no mesmo ponto.

O quadrirotor é considerado um corpo rígido no espaço que possui seis graus de liberdade (x,y,z,ϕ,θ,ψ), conforme Figura 3. O movimento em x, y e z é chamado de translacional e o movimento em ϕ,θ e ψ (leia-se, em inglês, *roll*, *pitch* e *yaw*, ou em português, rolagem, arfagem e guinada) é chamado de rotacional. Para se locomover em cada uma dessas dinâmicas, o quadrirotor é controlado através de quatro comandos que resultam da rotação dos quatro motores, estes são: torque no eixo x, torque no eixo y, torque no eixo z e *throttle*[1] (τ_ϕ,τ_θ,τ_ψ,T).

O comando de *throttle* é uma força resultante no eixo z que implica no movimento vertical de subida ou descida do veículo. Para esse movimento ocorrer todos os rotores devem aumentar ou diminuir a rotação na mesma proporção. Quando o empuxo aplicado pelos rotores em conjunto com as hélices supera a força peso, o veículo ganha altitude. Já quando a velocidade dos motores diminuem a ponto de ser superada pela força peso, o veículo perde altitude.

O comando de torque no eixo x implica no movimento de *roll* (ϕ) e está ligado à forma de atuação do par de motor 2-4 conforme a Figura 2. Ao diminuir a velocidade do

[1] Soma das forças de empuxo exercida pelos motores, mas será usado o nome em inglês por ser um termo bem conhecido em vários trabalhos.

motor 2 e aumentar a velocidade do motor 4 ocorre um movimento positivo de *roll*. Se a velocidade do motor 4 diminui e a velocidade do motor 2 aumenta ocorre um movimento negativo de *roll*.

Já o comando de torque no eixo y implica no movimento de *pitch* (θ) e ocorre de maneira similar ao movimento de *roll*. Ele é executado de acordo com a atuação dos motores 1-3 (Figura 2). Se a velocidade do motor 3 aumenta e a do motor 1 diminui, ocorre um movimento positivo de *pitch*. Caso o contrário, tem-se um movimento negativo de *pitch*.

Por fim, o torque no eixo z gera movimento de *yaw* (ψ), que é executado quando os motores que estão no mesmo eixo, ou motores que giram no mesmo sentido, tenham velocidades diferentes do outro par de motores que giram no sentido oposto, conforme visto na Figura 2. Isso ocorre devido à força de torque gerada pelos motores que será vista na próxima seção. Dessa forma existe uma força horizontal resultante capaz de girar o veículo em torno do seu eixo em sentido horário ou anti-horário de acordo com os pares de rotores que executam a maior velocidade.

2.1.3 Modelo matemático do quadrirotor

Dois métodos são comumente usados para encontrar o modelo matemático do quadrirotor. O método de Newton-Euler é utilizado em [21], nesse trabalho porém, será abordado o formalismo de Euler-Lagrange cujas equações podem ser obtidas através da análise de energia e trabalho [51]. A derivação será apresentada tendo como base o trabalho publicado em [37].

A Figura 3 apresenta os dois referenciais usados no modelo do quadrirotror. O referencial inercial, cujo centro é fixo na Terra, define a posição linear absoluta do quadrirotor $\boldsymbol{\xi}$. A posição angular, também chamada de atitude, é definida com relação ao referencial inercial pelos três ângulos de Euler $\boldsymbol{\eta}$. No vetor $\boldsymbol{q}$ estão contidas estas posições lineares e angulares referidas. A origem do referencial do corpo coincide com o centro de massa do veículo, e portanto, podem-se determinar as velocidades lineares $\mathbf{V}_{\mathrm{B}}$ e as velocidades angulares do veículo $\boldsymbol{\nu}$ neste referencial, com base nas equações apresentadas em [37]:

$$\boldsymbol{\xi} = \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \end{bmatrix}, \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \quad \boldsymbol{q} = \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix} \tag{2.1}$$

$$\mathbf{V}_{\mathrm{B}} = \begin{bmatrix} v_{\mathrm{x,B}} \\ v_{\mathrm{y,B}} \\ v_{\mathrm{z,B}} \end{bmatrix}, \quad \boldsymbol{\nu} = \begin{bmatrix} \mathrm{p} \\ \mathrm{q} \\ \mathrm{r} \end{bmatrix} \tag{2.2}$$

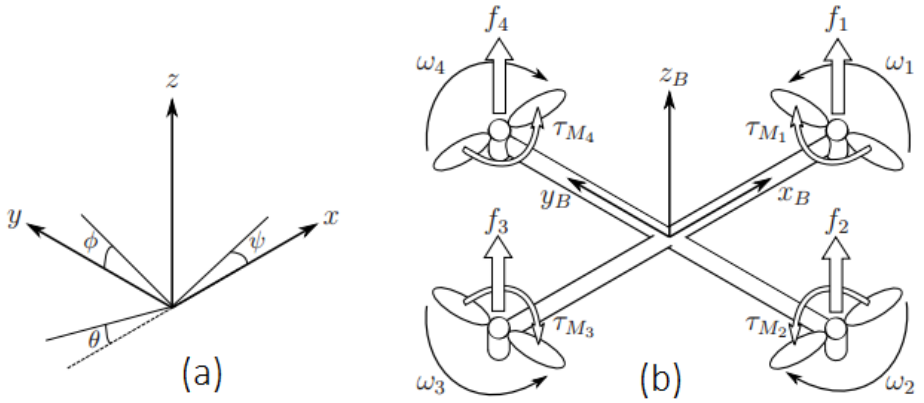

Figura 3 – (a)Referencial inercial e (b)referencial do corpo. Retirado de [37]

A matriz de rotação do referencial do corpo para o referencial inercial é:

$$\mathbf{R} = \begin{bmatrix} \mathrm{C}_\psi \mathrm{C}_\theta & \mathrm{C}_\psi \mathrm{S}_\theta \mathrm{S}_\phi - \mathrm{S}_\psi \mathrm{C}_\phi & \mathrm{C}_\psi \mathrm{S}_\theta \mathrm{C}_\phi + \mathrm{S}_\psi \mathrm{S}_\phi \\ \mathrm{S}_\psi \mathrm{C}_\theta & \mathrm{S}_\psi \mathrm{S}_\theta \mathrm{S}_\phi + \mathrm{C}_\psi \mathrm{C}_\phi & \mathrm{S}_\psi \mathrm{S}_\theta \mathrm{C}_\phi - \mathrm{C}_\psi \mathrm{S}_\phi \\ -\mathrm{S}_\psi & \mathrm{C}_\theta \mathrm{S}_\phi & \mathrm{C}_\theta \mathrm{C}_\phi \end{bmatrix} \tag{2.3}$$

em que $\mathrm{S}_x = sin(x)$ e $\mathrm{C}_x = cos(x)$.

Assume-se que o quadrirotor tem uma estrutura simétrica, e portanto, os seus momentos de inércia em torno de x,y,z são uma matriz diagonal $\mathrm{I}_{xx} = \mathrm{I}_{yy}$, uma vez que os braços estejam perfeitamente alinhados nos eixos x e y.

As velocidades angulares dos motores do veículo geram forças f_i na direção do eixo de rotação do motor, e geram também, torques τ_{M_i} ao redor do eixo dos motores:

$$f_i = kw_i^2, \quad \tau_{M_i} \approx bw_i^2 \tag{2.4}$$

em que k é o coeficiente de empuxo e b é o coeficiente de arrasto dos motores. A combinação das forças exercidas pelos quatro motores geram empuxo (*throttle*) T na direção do eixo z. Adicionalmente, o vetor $\boldsymbol{\tau_b}$ consiste nos torques em cada eixo do referencial do corpo, formalizando-se assim, as quatro forças que podem ser aplicadas ao corpo através dos motores:

$$T = k\sum_{i=1}^{4} w_i^2, \quad \boldsymbol{T}^B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{2.5}$$

$$\boldsymbol{\tau_b} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk(-w_2^2 + w_4^2) \\ lk(-w_1^2 + w_3^2) \\ \sum_{i=1}^{4} \tau_{M_i} \end{bmatrix} \tag{2.6}$$

em que l é a distância entre o motor e o centro de massa do corpo. Conforme visto anteriormente, torques no eixo x são gerados pelos motores 2 e 4, ao passo que torques em y são gerados pelos motores 1 e 3.

Para obter as equações nas dinâmicas mostradas na Figura 3, deve-se primeiramente obter o Lagrangiano $\mathcal{L}$, que é definido como a soma da energia translacional e rotacional menos a energia potencial, para cada uma das coordenadas generalizadas:

$$\mathcal{L}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{m}{2}\dot{\boldsymbol{\xi}}^{\boldsymbol{T}}\dot{\boldsymbol{\xi}} + \frac{1}{2}\boldsymbol{\nu}^{\boldsymbol{T}}\boldsymbol{I}\boldsymbol{\nu} - mgz \tag{2.7}$$

Em que m é a massa, g é a gravidade e $\boldsymbol{I}$ é a matriz dos momentos de inércia. Aplica-se então as equações de Euler-Lagrange para as forças externas e torques:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{\tau} \end{bmatrix} = \frac{d}{dt}\Big(\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}}\Big) - \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}} \tag{2.8}$$

O resultado da aplicação do método de Euler-Lagrange para as acelerações angulares do quadrirotor é dado por:

$$\ddot{\boldsymbol{\eta}} = \boldsymbol{J}^{-1}(\boldsymbol{\tau}_B - \boldsymbol{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}) \tag{2.9}$$

$\boldsymbol{J}$ é a matriz jacobiana que rotaciona um vetor de $\boldsymbol{\nu}$ para $\dot{\boldsymbol{\eta}}$ levando-se em conta os momentos de inércia para cada dinâmica, e $\boldsymbol{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})$ é termo de *Coriollis* que contêm as parcelas devidas aos efeitos giroscópicos e demais efeitos de natureza não-linear. Portanto, o modelo não-linear do quadrirotor pode ser escrito como:

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} \frac{\tau_\phi}{I_{xx}} + \frac{I_{yy} - I_{zz}}{I_{xx}}\dot{\psi}\dot{\theta} \\ \frac{\tau_\phi}{I_{yy}} + \frac{I_{zz} - I_{xx}}{I_{yy}}\dot{\psi}\dot{\phi} \\ \frac{\tau_\psi}{I_{zz}} + \frac{I_{xx} - I_{yy}}{I_{zz}}\dot{\theta}\dot{\phi} \\ (C_\psi S_\theta C_\phi + S_\psi S\phi)\frac{T}{m} \\ (S_\psi S_\theta C_\phi + S_\phi C\psi)\frac{T}{m} \\ -g + (C_\theta C\phi)\frac{T}{m} \end{bmatrix} \tag{2.10}$$

Para este trabalho, considera-se sempre a hipótese que os ângulos de atitude se mantêm pequenos (veiculo pairando sob o ar), e portanto, os efeitos não-lineares são

desprezados, obtendo-se então, o seguinte modelo para as acelerações angulares do veículo:

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{\tau_\phi}{I_{xx}} \\ \frac{\tau_\phi}{I_{yy}} \\ \frac{\tau_\psi}{I_{zz}} \end{bmatrix} \tag{2.11}$$

Por fim, para as acelerações lineares do quadrirotor, desconsiderando os efeitos aerodinâmicos como o arrasto, obtêm o seguinte modelo:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -g + (C_\theta C_\phi)\frac{T}{m} \end{bmatrix} \tag{2.12}$$

2.1.4 Modelo Linearizado

Assumindo que os ângulos de atitudes são bem pequenos (veículo pairando sob o ar), pode-se aproximar o modelo matemático do quadrirotor por um modelo linear. Aplicando a expansão de Taylor de primeira ordem em torno de um ponto de equilíbrio, o modelo discreto pode ser representado em espaço de estados conforme apresentado em [36]:

$$\sigma(k+1) = A\sigma(k) + Bu(k) \tag{2.13}$$

em que $\sigma = [x\ \dot{x}\ y\ \dot{y}\ z\ \dot{z}\ \phi\ \dot{\phi}\ \theta\ \dot{\theta}\ \psi\ \dot{\psi}]^T$ é o vetor de estados e $u = [w_1\ w_2\ w_3\ w_4]^T$ é o vetor de entrada (comandos). A é a matriz de estados e B é a matriz de entradas. O termo k vem do fato de que esta equação é discretizada no tempo de acordo com um período de amostragem τ_s, em torno de um ponto de equilíbrio (veículo pairando) [34].

2.1.5 Estimador e sensores

Para estimar o vetor de estados do sistema é necessário sensores de medida inercial, também conhecido como *Inertial Measurement Unit* (*IMU*). São eles o giroscópio, o acelerômetro e o magnetômetro (Figura 4).

O girômetro ou giroscópio é um sensor de extrema importância para o funcionamento do sistema, pois este é responsável pelas medidas de velocidade angular, a partir das quais podem ser estimadas as acelerações angulares[23]. Sua utilização é necessária para detectar as mudanças nas rotações nos eixos de *roll*, *pitch* e *yaw*, portanto os sensores de navegação da *IMU* utilizam 3 giroscópios, um para cada eixo desejado (conforme

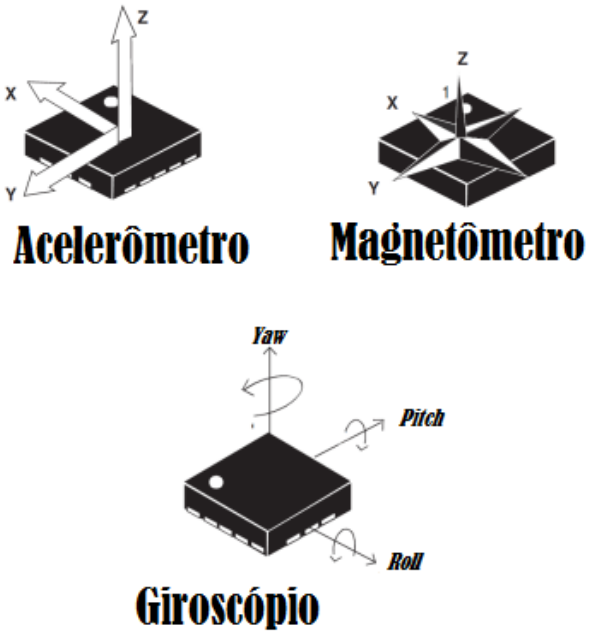

Figura 4 – Principais sensores da *IMU*

Figura 4). Esse sensor é de fundamental importância para estimar o vetor de estados do veículo.

O acelerômetro é outro sensor fundamental para o quadrirotor, pois detecta a aceleração ao longo do eixo no qual foi inserido. A *IMU* utiliza 3 acelerômetros, um para cada eixo assim como o giroscópio, tornando possível medir acelerações nos eixos (x,y,z).

O magnetômetro funciona de maneira à medir os campos magnéticos. Com esse sensor, é possível conseguir uma referência robusta de orientação (norte, sul, leste, oeste), e portanto a *IMU* utiliza este sensor auxiliar na estimação do ângulo de *yaw*.

As medidas isoladas de cada um desses sensores não são suficientes para fornecer os 12 estados do veículo mencionados anteriormente. Primeiramente, o giroscópio não obtêm os ângulos de Euler diretamente, e sim as velocidades nestes ângulos, portanto existe um erro que é acumulado com o tempo no cálculo dos estados (ângulos de Euler) atuais. As medidas do acelerômetro não são estáveis quando as acelerações são elevadas, e não é possível avaliar a medida de aceleração em *yaw* através deste sensor. Por fim, o magnetômetro está sujeito a vários tipos de interferência magnética, além de precisar de uma grande estimativa de inclinação para fornecer dados corretos [45].

Portanto é necessário um método para estimar esses estados de uma maneira precisa, e isso é feito através de um observador de estados que realiza fusão sensorial. Fusão sensorial é a combinação de dados sensoriais ou de dados derivados de sensores cuja informação resultante é melhor do que seria possível caso cada um destes sejam usados individualmente [24].

A fusão sensorial promove os ajustes necessários para estimar corretamente os estados [38]. Os dados dos sensores são filtrados e corrigidos de acordo com o método escolhido que melhor se adequa ao sistema em que se está trabalhando.

A estratégia mais famosa para fusão sensorial em veículos do tipo quadrirotor é o filtro de *Kalman* extendido, uma versão não-linear do filtro de *Kalman*. Este modelo é um conjunto de equações que se dividem em duas categorias: equações preditivas e equações corretivas. O filtro de Kalman extendido produz estimativas estatísticamente ótimas quando utilizado em um sistema com modelo linear e é bastante utilizado apesar da sua pesada carga computacional [45].

O filtro de *Kalman* extendido é a solução que fornece todos os estados necessários de uma maneira precisa. A plataforma escolhida para este trabalho possui um filtro de *Kalman* extendido implementado e isto será visto com mais detalhe no decorrer deste trabalho.

2.1.6 Técnicas de controle para quadrirotor

Para que um quadrirotor possa voar de maneira adequada, existe a necessidade de controlar os seus estados através de algoritmos computacionais. Quando um comando para execução de algum movimento (movimento de 5° em *pitch* por exemplo), é enviado do transmissor (rádio ou de um *software* embarcado) para a aeronave, o sistema de controle é responsável por ler esse sinal e modificar a rotação dos motores com a precisão necessária para que o movimento seja de acordo com o esperado. O controlador deve estar calibrado para executar os seus passos no tempo correto, caso contrário o vôo do quadricóptero estará comprometido.

Apesar do quadrirotor possuir uma mecânica relativamente simples quando comparada com outros veículos aéreos, seu controle não é trivial devido a sua natureza subatuada, modelo altamente não-linear e dinâmicas acopladas [55]. Sendo assim, a teoria de controle é de fundamental importância para a elaboração das estratégias de controle para o quadrirotor.

Diversas estratégias de controle já foram utilizadas para quadrirrotores. O controlador proporcional-integral-derivativo (PID) é um controlador bem consolidado na indústria e bastante robusto. O regulador linear quadrático (LQR) é um controlador ótimo que possui eficiente rejeição à pertubações [55]. O controlador preditivo baseado em modelo (MPC) é um controlador com restrições que apresenta bastante potencial para lidar com problemas de segurança e limitação dos atuadores [36], porém não será objeto de estudo neste trabalho.

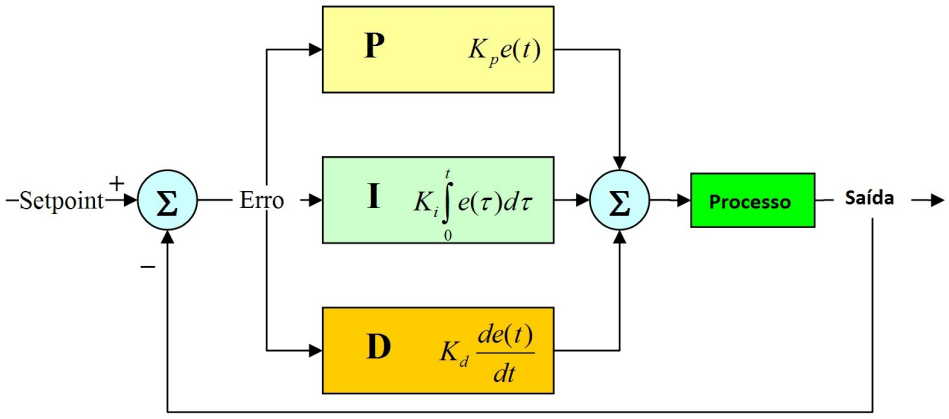

Figura 5 – Controle PID

2.1.6.1 Controlador *PID*

Uma das técnicas de controle mais utilizadas na indústria é o controlador *PID* (*Proportional Integral Derivative*) devido a sua estrutura e *tunning* (ajuste de ganhos) relativamente fácil quando comparada com outras técnicas . Esta técnica consiste em obter o erro - através do seu mecanismo de controle com realimentação - que é a diferença entre o ponto em que o sistema está atualmente e o ponto desejado (*set point*), e assim, buscar minimizar este erro modificando os atuadores do sistema através da alteração dos ganhos [30], como pode ser visto na Figura 5.

Os três diferentes ganhos envolvidos nesta técnica - P, I e D - são descritos em termos do tempo, sendo o erro presente dependendente de P, o erro passado dependente de I e o erro futuro dependente de D [43]. A alteração de cada um desses ganhos promove uma mudança nas propriedades da resposta natural do sistema (Figura 6), apresentadas abaixo com base em [4]:

- *Rise time*: tempo de subida. Tempo que o sinal de interesse no sistema leva para ir de 10% à 90% (procentagens típicas) do valor final.
- *Overshoot*: é o valor que ultrapassa à resposta final.
- *Settling time*: tempo de estabilização. Tempo que o sistema leva para chegar à 5% (porcentagem típica) do valor final.
- Erro de estado estacionário: diferença final entre a resposta desejada e o *set point*.

O termo P, também chamado de coeficiente de ganho proporcional, é o ganho mais importante pois determinará quão reativo é o sistema à variações. Ao aumentar-se o ganho P o sistema responde mais rápido (redução do *rise time*), porém o *overshoot* e o erro de estado estacionário também aumentam o que pode causar instabilidade (Figura 7).

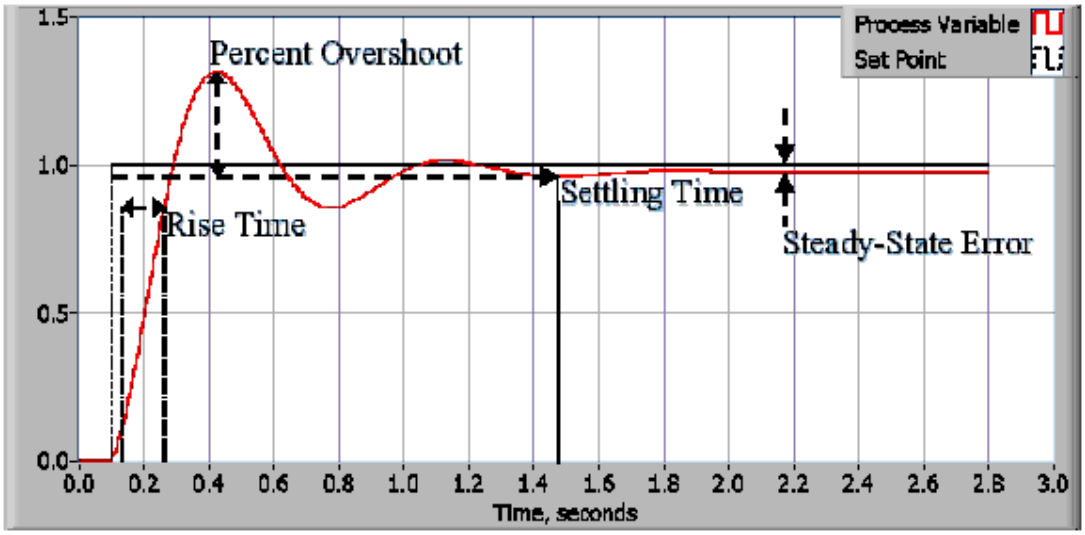

Figura 6 – Respostas típicas de um controlador *PID*. Retirado de [4].

Aumento do ganho	*Rise time*	*Overshoot*	*Settling Time*	Erro de estado estacionário
P	↓	↑	Pequena mudança	↓
I	↓	↑	↑	Grande diminuição
D	Pequena mudança	↓	↓	Pequena mudança

Figura 7 – Figura indicando a alteração na resposta transiente do sistema com o aumento dos ganhos *P*, *I* e *D*. Adaptado de [9].

O ganho *I*, conhecido como o termo integral, é importante para tornar o sistema mais preciso devido ao erro passado acumulado. O aumento do ganho *I* proporciona uma redução no erro de estado estacionário fazendo o sistema estabilizar mais perto do ponto desejado. Em contrapartida, o aumento do termo *I* causa um aumento do *overshoot* e um aumento no tempo que o sistema leva para estabilizar (*settling time*) (Figura 7).

Por fim, o ganho derivativo *D*, faz com que o sistema aumente a velocidade de reação de acordo com a taxa de variação do erro. Um aumento no ganho derivativo proporciona uma redução no *overshoot* e no *settling time*, mas pode ocasionar variações no erro de estado estácionário e no tempo de subida caso não ajustado corretamente (Figura 7).

No caso do quadrirotor, os controladores *PID* são usados para o controle em baixo nível do veículo, isto é, controle de posição (x,y,z) e controle angular (atitude) (*roll*, *pitch* e *yaw*). Ao estabelecer-se o ponto de operação [2] o controlador executará os *loops* de controle até que essa condição seja alcançada.

[2] Um exemplo de ponto de operação comum para o quadrirotor é aquele em que todos os ângulos desejados são zero

2.1.6.2 Controlador LQR

A técnica de controle conhecida como Regulador Linear Quadrático, ou em inglês, *Linear Quadratic Regulator* é uma técnica de controle ótimo, cujo objetivo é operar em sistemas dinâmicos com um custo mínimo. Esta técnica é usada em um projeto de um quadrirotor *indoor* em [20].

Dado um modelo linear no espaço de estados:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{2.14}$$

em que $\mathbf{A}$ é matriz de estados, $\mathbf{B}$ é a matriz de entradas, $\mathbf{u}$(t) é vetor de entradas e $\mathbf{x}$(t) é vetor de estados, o *LQR* resultará em uma lei de controle:

$$\mathbf{u}(t) = -\mathbf{K}\mathbf{x}(t) \tag{2.15}$$

em que K é a matriz constante de ganhos, que minimiza a função custo:

$$J = \int_0^\infty (\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u})dt \tag{2.16}$$

As matrizes $\mathbf{Q}$ e $\mathbf{R}$ são responsáveis por penalizar ou por amplificar os estados e as entradas do sistema, respectivamente. Para encontrar a matriz $\mathbf{K}$, a lei de controle ótimo é dada por:

$$\mathbf{u}(t) = -\mathbf{K}\mathbf{x}(t) = -\mathbf{R}^{-1}\mathbf{B}^T\mathbf{P}\mathbf{x}(t) \tag{2.17}$$

sendo que a matriz $\mathbf{P}$ é encontrada através da solução da equação de *Riccati*:

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} - \mathbf{P}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\mathbf{P} + \mathbf{Q} = 0 \tag{2.18}$$

O *LQR* é um controlador cujo objetivo é minimizar uma função de custo definida por quem está projetando o controle. Se por um lado ele lida melhores com sistemas com múltiplas entradas e múltiplas saídas (*MIMO* em inglês) [22], por outro lado possui certas dificuldades na definição da função custo e nas matrizes de ponderação.

2.2 Arquitetura do quadrirotor

A arquitetura de um sistema quadrirotor pode variar dependendo das características do veículo e que funções este deve exercer. Entretanto, de uma maneira geral a arquitetura pode ser representada de acordo com a Figura 8. A *Main Board* é a placa de

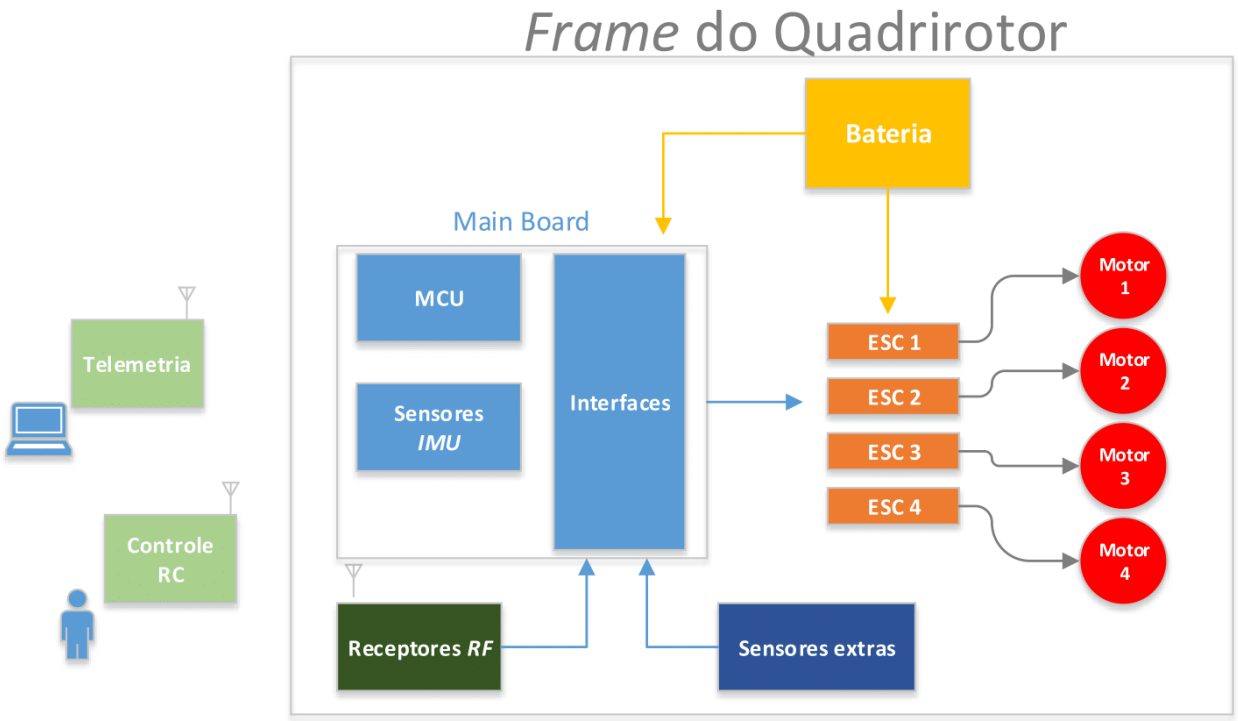

Figura 8 – Arquitetura do sistema quadrirotor

controle e processamento, também chamada de *flight controller*, que possui o microcontrolador principal (*MCU*) e é responsável por gerenciar a comunicação entre os demais dispositivos, conversando com estes através das interfaces. O sensores *IMU* também estão integrados na *Main Board*[3]. O receptor de rádio-frequência (*RF*), representa os receptores de sinal de rádio e de telemetria, e os sensores externos, qualquer outros sensor que não esteja fixado na *Main Board*.

Electronic Speed Controllers (*ESCs*) são os controladores de velocidade dos motores. A alimentação dos motores, sensores e demais dispositivos é fornecida por uma bateria. Os motores possuem hélices acopladas que permitem que o veículo execute um vôo.

Por fim, tem-se o controle de rádio operado por um humano que envia sinais de comando para o veículo, e um computador em terra executando um *software* de missão que se comunica com o veículo através de um dispositivo de telemetria. Esses dispositivos estão fora do *frame* do quadrirotor.

Esta é um arquitetura genérica, que, apesar de não ser uma regra para os quadrirotores, é bastante adotada atualmente. Os detalhes dos principais componentes serão vistos nesta seção. Demais informações podem ser encontradas nos Apêndices.

[3] Existem quadrirotores em que os sensores *IMU* se localizam fora da placa controladora principal.

2.2.1 Placa controladora de vôo (*Main Board*)

A *Main Board*, ou *Flight Controller*, é uma placa com os principais dispositivos eletrônicos embarcados no veículo. Nela estão os sensores de navegação, as interfaces de comunicação, e o processador principal. Estes são os principais componentes da placa, que pode possuir diversos outros dispositivos de acordo com a necessidade do veículo.

Toda placa controladora de vôo possui um processador principal, que está contido geralmente dentro de um microcontrolador *MCU*. Em sistemas robóticos é comum ver o uso de microcontroladores, pois estes possuem além de um processador, memórias internas, terminais de entrada e saída (*E/S*, ou em inglês, *I/O*), além de outras interfaces de comunicação. Abaixo, têm-se uma lista de interfaces comuns de serem encontradas em microcontroladores modernos:

- *UART, I2C, SPI*: protocolos padrões de comunicação serial que possibilita a comunicação com computadores externos (*offboard*), sensores, e demais dispositivos que utilizam-se destes protocolos.
- *PWM*: a maior parte dos quadrirotores atuais enviam sinais de controle utilizando-se de *Pulse Width Modulation* (*PWM*), portanto é necessário que o *MCU* forneça pelo menos quatro saídas PWM.
- *GPIO*: são os pinos de propósito geral do *MCU*, que possibilita o desenvolvedor adicionar funcionalidades conforme desejadas.
- *A/D*: conversores A/D são essenciais para leitura de sensores analógicos
- *JTAG, USB*: portas padrão para *debug* e gravação do *firmware* no *MCU*.

O *MCU* executa a aplicação embarcada com os algoritmos de processamento em tempo-real, leitura dos sensores e envio de sinais de comando para os respectivos motores. O *MCU* deve ser capaz de executar os algoritmos de controle de trajetória, posição e atitude, realizar operações de *E/S* com demais dispostivos, sendo portanto, importante a escolha de um *MCU* correto para a aplicação. Quanto mais complexo a arquitetura de *software* do sistema for, maior capacidade de processamento será exigida do *MCU*. As interfaces de comunicação que o *MCU* possui são muito importantes, pois definem como os dispositivos externos devem se comunicar ele.

A *Main Board* possui, além do microcontrolador, os sensores de navegação (*IMU*) para efetuar vôo. São estes o acelerômetro, girômetro e magnetômetro (apresentados na Seção 2.1.4). Existem arquiteturas porém, que integram mais sensores na *Main Board*.

As interfaces da *Main Board* são necessárias para a comunicação desta com os demais dispositivos e sensores. Na maioria dos casos elas são fornecidas diretamente pelo

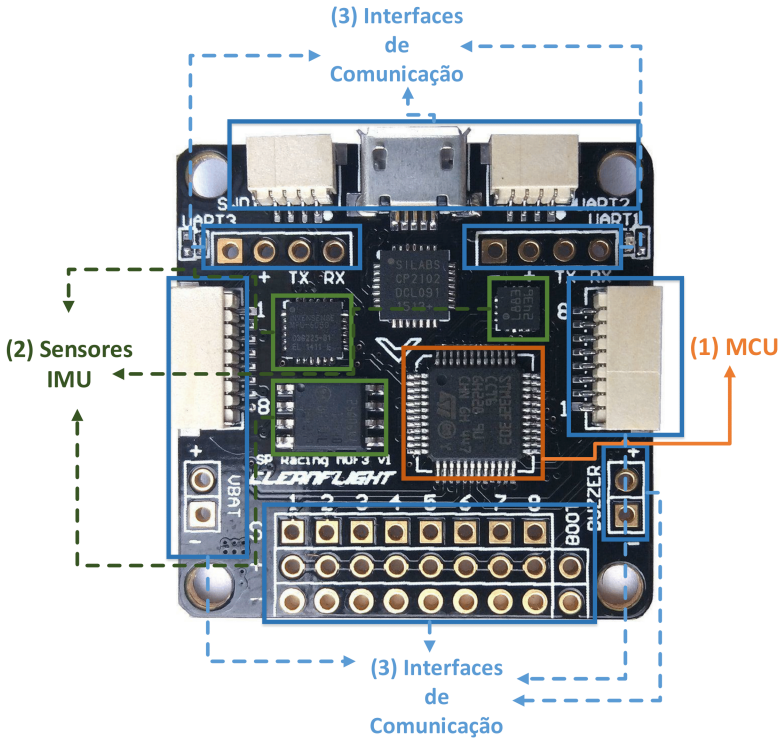

Figura 9 – *Main Board* comercial conhecida como *CleanFlight*. Fonte da imagem desconhecida.

microcontrolador, porém, em alguns casos são providas por dispositivos intermediários. As principais interfaces para comunicação foram apresentadas na seção que se introduziu o microcontrolador.

A Figura 9 mostra um exemplo de placa de controle de vôo comercial. Pode-se ver na imagem em (1) o *MCU*, em (2) os sensores de navegação e em (3) inúmeras interfaces de comunicação.

2.2.1.1 Sensores extras e Receptor *RF*

Os sensores extras representam todos os sensores que não estão diretamente contido na *Main Board* mas estão no *frame* do quadrirotor. Existem uma variedade de sensores utilizados para este tipo de veículo nos dias de hoje, o que está diretamente relacionado ao tipo de aplicação em que o quadrirotor em questão é construído.

Um sensor externo famoso, presente em quase todos os *VANTs* modernos é o sensor de posicionamento global *GPS* (exemplo na Figura 10), utilizado para vôos autônomos. Este sensor é muito útil para vôos *outdoor* (vôos em ambiente aberto). Uma das utilidades deste sensor é para controle do quadrirotor através de *softwares* de navegação, onde pode-se enviar posições de *GPS* reais (*waypoints)* para o veículo alcançar. O veículo compara sua posição de *GPS* com a posição fornecida pelo *software*, e assim o veículo consegue, teoricamente, realizar vôo para qualquer posição global.

Figura 10 – Sensor *GPS* da empresa *3DR*. Este sensor acompanha o *kit* da *Pixhawk*.

Outro sensor bastante utilizado é o barômetro. Este sensor trabalha medindo as mudanças de pressão atmosférica, sendo utilizado, portanto, para realizar medições de altura do veículo. Ainda, com a mesma finalidade, tem-se o sonar, sensor utilizado para medir altura e em aplicações em que se deseja realizar trajetórias escapando de obstáculos. Sensores de fluxo ótico, sensores *LIDAR* e câmeras são também utilizados para este propósito.

Os receptores de radio-frequência (*RF*) são componentes encontrados em quase todos os quadrirotores modernos. Estes representam os dispositivos que recebem os sinais de rádio (piloto), e também, receptores de sinais de dispositivos de telemetria. Estes dispositivos captam os sinais enviados em uma certa frequência, e se comunicam com o microcontrolador da *Main Board*. Estes dispositivos são de fundamental importância, pois comandos de controle e informações cruciais são enviadas durante vôo através destes. Os receptores de sinais de telemetria, não só recebem, como também enviam sinais com dados.

A Figura 12 mostra as interfaces de comunicação com a *Pixhawk*. Ela dispõe de diversos padrões de comunicação já citados (como *UART*, *I2C*), e ainda possui *LEDs* para informar o condutor sobre eventuais ocorrências. Ela fornece ainda, comunicação USB, e espaço para cartão SD. Mais detalhes sobre a *Pixhawk* serão dados na Seção 3.3.5.

2.2.2 *Electronic Speed Controller*

O *Electronic Speed Controller* (*ESC*) é um controlador de velocidade dos motores do veículo, sendo que cada motor precisa do seu próprio (exemplo de *ESC* na Figura 13). A necessidade do *ESC* existe devido ao tipo de motor utilizado em quadrirotores: o motor *brushless DC*. Devido à sua característica trifásica, este tipo de motor possui bobinas que precisam ser acionadas na sequência correta para que o movimento faça sentido.

O *ESC* funciona da seguinte maneira: primeiro ele recebe os sinais de controle

Figura 11 – Típico transmissor *RC*. Adaptado de [27]

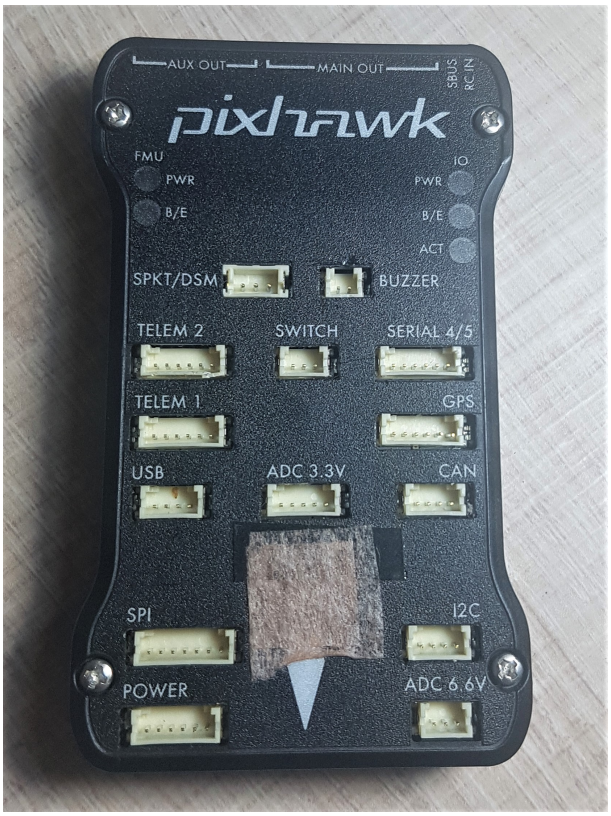

Figura 12 – Interfaces de comunicação da *Pixhawk*

Figura 13 – *ESC* da empresa *DJI*. Adaptado de [3]

vindos da *flight controller* (geralmente um sinal modulado por pulsos, conhecido como *PWM*), e então, converte esse sinal em potência (tensão e corrente) provida da bateria para os motores. A direção de rotação também é controlada pelo *ESC*. Devido ao fato do *ESC* interagir com a bateria e os motores, a sua escolha é dependente das especificações destes. Alguns dos fatores a se verificar ao escolher um *ESC* de acordo com [42] são :

- Corrente máxima que o motor necessita em máxima rotação
- Protocolo dos sinais de entrada (se *PWM* ou outro)
- Suporte para quantidade de células da bateria a ser utilizada
- Peso e tamanho

2.2.2.1 Motores

Motores *brushless DC* (Figura 14), ou motores de corrente contínua sem escovas são utilizados em quadrirotores devido a sua maior durabilidade, eficiência e reduzida necessidade de manutenção quando comparado com os motores *DC* com escova (*brushed DC*).

Os principais fatores que devem ser conhecidos para a correta escolha do motor são a tensão, corrente, propulsão, potência, eficiência e velocidade. Portanto deve-se saber qual o peso da estrutura que irá voar para que os motores possam trabalhar com uma certa "folga"e não consumam toda a energia da bateria rapidamente. O peso é também, fundamental para calcular a propulsão necessária para a escolha do motor. Uma fórmula geralmente usada para encontrar a propulsão é multiplicar o peso total da estrutura por dois, e dividir pelo número de motores (quatro para quadrirrotores) [15].

Figura 14 – *Motor DC* da *DJI*, bem comum em kits da *Pixhawk*

Os motores *brushless* são classificados em *Kv*, unidade de medida que relaciona rotações por minuto (*RPM*) com a tensão necessária (em *Volts*). Geralmente motores com elevado *Kv* são usados em estruturas mais leves, em sistemas que priorizam velocidade ou rápidas acrobacias. Motores com *Kv* menores são geralmente utilizados em estruturas mais pesadas como quadricópteros que precisam carregar equipamentos de imagem. Os motores também são classificados quanto ao número de pólos que pode variar de dois até quatorze. Motores menores que para quadrirotores mais rápidos possuem menor número de pólos ao passo que motores maiores para carregar estruturas pesadas possuem elevado número de pólos [15].

Uma combinação adequada do motor com o tamanho das hélices a serem utilizadas é fundamental, pois isso influenciará na rotação e, consequentemente, no desempenho do sistema. É desejável que um motor rápido seja combinado com uma hélice mais curta, e que um motor mais lento que carregue peso seja combinado com uma hélice de maior comprimento.

2.2.3 Fornecimento de energia

A bateria é a parte responsável por fornecer energia elétrica para alimentar os motores do quadricóptero. As baterias do tipo polímero de lítio (*LiPo*, Figura 15) são as mais usadas para esse propósito por conseguirem acumular uma maior densidade de energia comparada com outros tipos.

Segundo [6] as baterias são escolhidas de acordo com a necessidade do motor e a especificação do *ESC*. O conhecimento da quantidade de corrente que será drenada por vôo é essencial, pois assim uma escolha correta de capacidade de bateria pode ser feita. O número de células usadas deve ser compatível com a quantidade de células em que o *ESC* trabalha.

Os quadrirotores utilizam, normalmente, uma bateria, porém existem aqueles que utilizam duas para garantir uma melhor segurança caso uma falhe durante a missão, e

Figura 15 – Bateria de *LiPo* utilizada em *VANTs*

também para garantir uma maior autonomia, porém, ao preço de um custo mais elevado, maior dificuldade de montagem da estrutura e aumento do peso. Portanto o *tradeoff* que deve ser avaliado com cautela.

2.3 Conceitos de sistemas embarcados

O quadrirotor possui um sistema eletrônico embarcado, apresentado na seção anterior, que possui um microcontrolador responsável por executar as aplicações de *software* que permitem o controle do veículo. Nesta seção serão apresentado importantes conceitos que fundamentaram este projeto: o modelo de um sistema embarcado e sistemas operacionais de tempo-real.

2.3.1 Modelo de um sistema embarcado

O modelo completo de um sistema embarcado pode ser representado por três camadas de acordo com a Figura 16: camada de *hardware*, camada de *software* do sistema e a camada de aplicação. Interno à camada de *software* do sistema, existem três camadas importantes que são: camada de *device drivers*, camada do *sistema operacional* e camada de *middleware* [40].

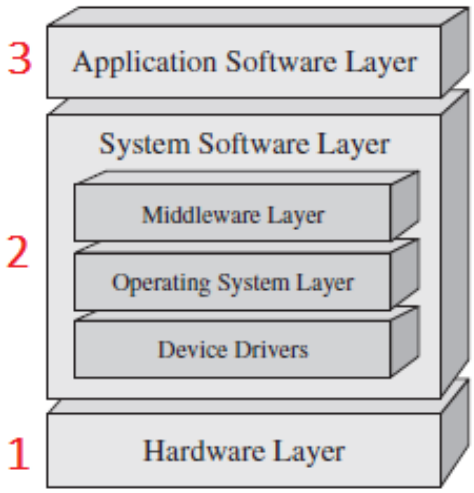

Figura 16 – Modelo de um sistema embarcado: **1** - camada de *hardware*; **2** - camada de *software* do sistema e **3** - camada de aplicação. Adaptado de [40].

A camada de *hardware* é a camada física do sistema que contêm todos os dispositivos mecânicos, elétricos e químicos. Os resistores, capacitores, diodos, transistores, e demais dispositivos eletrônicos discretos estão nesta camada. Mais importante ainda, nesta camada estão contidos o processador, responsável por executar o *software*, as memórias e os barramentos de dados. Esta é a camada mais importante, pois sem ela as demais camadas não existem.

A camada de *software* do sistema possui inúmeras responsabilidades, o que inclui gerenciar aplicações, gerenciar *hardware* e gerenciar memória. Esta camada possui três camadas internas importantes, sendo que estas camadas podem ser implementadas de

maneiras diferentes, e que o modelo apresentado aqui é apenas uma maneira de genereliźá-las.

A camada de *device drivers* é a camada mais abaixo da camada de *software* do sistema. Esta camada é de fundamental importância pois ela possui as aplicações de *software* básico (baixo-nível), chamadas de *drivers*, que inicializam e gerenciam o acesso aos dispositivos de *hardware*. Todo sistema que necessita de um *software* deve possuir *drivers* na sua camada de *software* do sistema. Para uma aplicação embarcada, cada dispositivo deve possui um *driver* desenvolvido para aquela plataforma (processador). Para ilustrar isto, têm se como exemplo um acelerômetro: para cada tipo de microcontrolador ou placa em que este será usado, é necessário o desenvolvimento de um *driver* para este dispositivo. Apesar de existirem *drivers* genéricos, de maneira geral, a maioria dos *drivers* são específicos da arquitutura.

A camada de sistema operacional, vêm logo acima da camada de *device drivers*. Quando o sistema embarcado possui um sistema operacional, esta é a camada em que este se localiza. O sistema operacional é uma série de bibliotecas e aplicações que provê abstrações para as demais aplicações em alto nível serem menos dependentes do *hardware*, fazendo com que o desenvolvimento destas aplicações seja facilitado. O sistema operacional também gerencia os recursos de *software* e *hardware* para garantir que todo o sistema opere de forma confiável e eficiente [40]. A seguir, têm-se uma lista das principais funcionalidades de um sistema operacional em um sistema embarcado:

- Gerenciamento de processos: como as aplicações (processos) são vistas pelo sistema operacional e como gerenciar suas ações. Como por exemplo, gerenciar as múltiplas interrupções geradas por um processo.
- Gerenciamento de memória: como a memória é compartilhada por múltiplos processos, é necessário um gerenciamento do uso dela por parte do sistema operacional. Um processo não pode, por exemplo, alocar páginas de memória que está sendo usado por outro processo, e o sistema operacional é quem deve gerenciar isto.
- Gerenciamento de recursos de *E/S*: dispositivos de entrada e saída são compartilhados entre as aplicações, e o sistema operacional é responsável por gerenciar esses recursos.

A camada de *middleware*, que está acima do sistema operacional, é uma camada onde está todo tipo de aplicação que não é o sistema operacional, um *driver* ou uma aplicação de *software* [40]. *Middlewares* são aplicações que intermediam serviços entre outras aplicações, são uma espécie de extensão de alguma funcionalidade que é bastante utilizada no sistema. Exemplos de funcionalidades que *middlewares* implementam são serviços de mensagens e comunicação.

Por fim, situada no topo, a camada de aplicação é onde todas as aplicações de *software* são executadas. Estas aplicações, também chamada de processos, dependem do sistema operacional para serem executadas. Conforme ressaltado em [40], esta camada define que tipo de sistema embarcado o dispositivo é, pois as aplicações em alto-nível representam o propósito daquele sistema e interagem com o usuário.

2.3.2 Sistemas Operacionais de Tempo-Real

Sistemas operacionais de tempo-real, ou em inglês, *Real-Time Operating Systems* (*RTOS*), são sistemas operacionais projetados para diversas finalidades, sendo que uma das suas importantes aplicações é para resposta à eventos e sistemas de controle de malha fechada [28]. Os sistemas operacionais de tempo-real têm como objetivo gerenciar as aplicações de maneira determinística, isto é, cumprindo as entregas (*deadlines*) dentro das restrições de tempo do sistema.

Os *RTOS* podem ser separados em duas classificações: *soft real-time* e *hard real-time* [54]. *RTOS* na categoria de *soft real-time*, são sistemas operacionais que funcionam cumprindo *deadlines*, mas a falha não compromete o sistema. Um exemplo é o *RTOS* em um celular *touchscreen* moderno: quando o sistema está executando muitos aplicativos e, portanto sobrecarregado, a tela do celular pode congelar durante alguns instantes e até irritar o usuário, entretanto, esta falha não compromete o sistema. Já um RTOS *hard real-time* executa em sistemas em que o cumprimento de *deadlines* deve ser respeitado obrigatoriamente, como em um sistema de um *VANT*, por exemplo. Se a unidade de processamento do *VANT* não calcular as velocidades dos motores dentro de um prazo previamente estabelecido, o sistema irá falhar e o veículo poderá ser danificado.

A principal diferença entre um *RTOS* e um sistema operacional de propósito geral (*Linux*, *Windows* e *Mac OS* por exemplo), é que este é projetado para executar uma variedade de aplicações de maneira simultânea, atribuindo tempo de processamento para todas elas. Já o *RTOS* possui uma capacidade de priorizar aplicações[4], assim uma aplicação crítica pode ter mais tempo de processamento disponível, mesmo que uma aplicação esteja esperando há mais tempo para ser executada (uma vez que ela tenha prioridade baixa). Esta característica permite que as aplicações em um *RTOS* tenham respostas previsíveis.

Os *RTOS* são utilizados em sistemas embarcados que executam aplicações que esperam um comportamento determinístico do sistema, como um controlador *PID* por exemplo. Determinismo é a medida de consistência do intervalo de tempo especificado entre os eventos [28]. As maneiras mais comuns de se medir o determinismo é através da latência e do *jitter* (Figura 17).

[4] É possível encontrar mais detalhes sobre o o funcionamento do escalonador nos Apêndices.

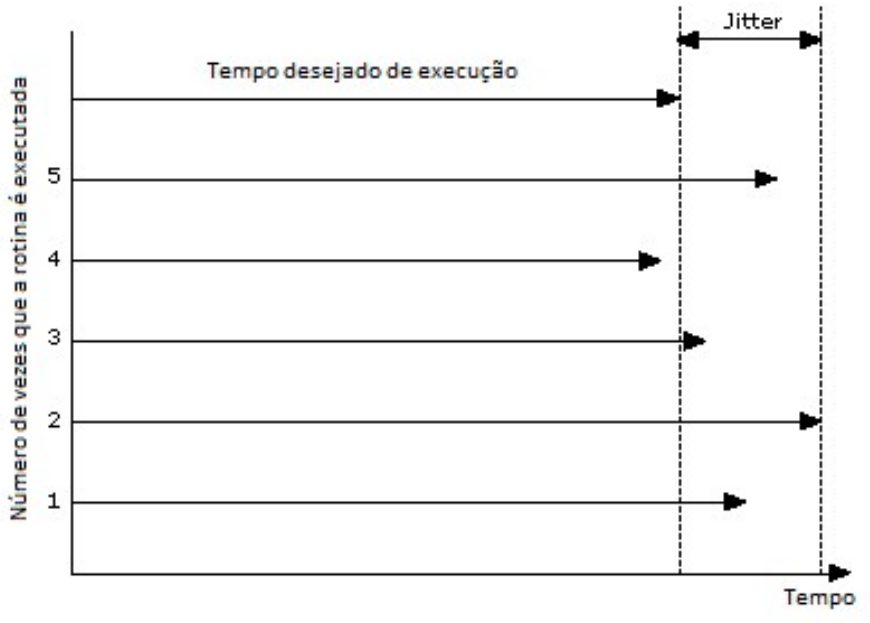

Figura 17 – Representação do *jitter*

A latência é o tempo que o sistema demora para responder. Se uma aplicação deseja ler os dados de um sensor, o tempo de espera até que esses dados estejam disponíveis é a latência. O *jitter* é diferença entre o tempo de execução desejado para uma rotina e o tempo em que ela de fato executa. Para um *RTOS*, essas duas métricas são extremamente importantes, e tanto a latência quanto o *jitter* devem ser pequenos e limitados.

3 *PX4 Autopilot*

Neste capítulo é apresentado incialmente uma revisão sobre a arquitetura do *firmware PX4*, [1] que encapsula um sistema operacional, um *middleware* para comunicação e camada de aplicação. É apresentado também neste capítulo algumas características do *hardware Pixhawk*, desenvolvido pelos mesmos fundadores da *PX4*. Por fim, uma breve introdução sobre os simuladores disponíveis para uso com a *PX4* é apresentada.

3.1 Introdução ao *firmware PX4*

O *firmware PX4* é uma solução para controle de vôo desenvolvido e mantido por múltiplos desenvolvedores ao redor do mundo, e executa em múltiplos *hardwares* diferentes. Por estar sob a licensa *BSD* - o que implica que todo o seu código é aberto e pode ser modificado por qualquer pessoa, - a *PX4* é extensivamente usada por diversos pesquisadores, empresas e hobistas.

Para entender o funcionamento da arquitetura da *PX4* deve-se ter em mente que esta consiste em três camadas principais: a PX4 *flight stack*, um *software* para realizar vôos autônomos, o *PX4 middleware*, que é um *middleware* para comunicação que suporta qualquer tipo de robô autônomo e um sistema operacional de tempo-real *NuttX* (Figura 18).

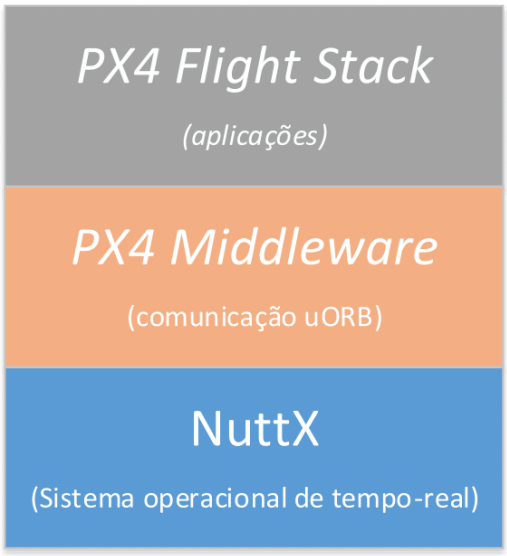

Figura 18 – *Firmware PX4*

[1] *Firmware* é um *software* embarcado em um dispositivo de *hardware*.

A *PX4 flight stack* é uma implementação de *software* de alto nível que é executada em cima da camada do *PX4 middleware*, porém, uma existe perfeitamente sem a outra [2]. A camada do *middleware* é executada em cima do sistema operacional. Esta é uma visão bem simplificada, e os detalhes serão apresentados no decorrer deste capítulo.

3.2 Critérios de desenvolvimento

Para desenvolver *software* eficientemente para sistemas embarcados que exigem controle em tempo real, como é caso de um quadrirotor, é necessário adotar alguns critérios. Diversos sensores precisam ser lidos com uma baixa latência e sinais de controles devem ser enviados respeitando as restrições temporais impostas pela estratégia de controle, portanto as interfaces de baixo nível como *I2C*, *SPI*, *CAN* e saídas *PWM* são desenvolvidas respeitando os requerimentos [39].

A Figura 19 mostra um exemplo de como o modelo dinâmico de um veículo (um *VANT* ou um robô) possui diferentes laços aninhados de controle, cada um com diferentes restrições de tempo. Pode-se ver na parte superior da imagem que os pontos de navegação e geração das linhas (ou curvas) de trajetória possui uma restrição de tempo cerca de dez vezes maior quando comparada com a geração das posições de controle, e centenas de milhares de vezes em relação a geração de sinais de controle para os atuadores. A importância de saber as restrições de tempo para cada laço no controle de um veículo está no fato de que pode-se escolher plataformas diferentes para cada um destes, e portanto, criar um sistema modularizado e flexível.

Os outros critérios estabelecidos no desenvolvimento para a *PX4* citado pelo autor são a reutilização e a interoperabilidade. A primeira diz respeito principalmente a reutilização de módulos do sistema por outras pessoas e pesquisadores. A *PX4* possui um alto nível de reusabilidade permitindo que diferentes aplicações e até mesmo *hardware* possam ser instalados sem ter que modificar a arquitetura do sistema. Já a interoperabilidade diz respeito à compatibilidade e integração que a *PX4* oferece aos usuários. Estes podem escrever seus códigos tanto no microcontrolador (*hardware* do sistema) quanto em um sistema operacional *Linux*, permitindo que o usuário possa efetuar testes como *software-in-the-loop* no seu computador.

3.3 Arquitetura do *firmware PX4*

A arquitetura do *firmware* da *PX4* foi introduzido anteriomente, e será apresentado com mais detalhes nesta seção. A Figura 20 apresenta as principais camada desta

[2] Um exemplo de aplicação em que um não depende do outro é um sistema em que o piloto automático *Ardupilot Mega APM* é executado em cima do *PX4 middleware*.

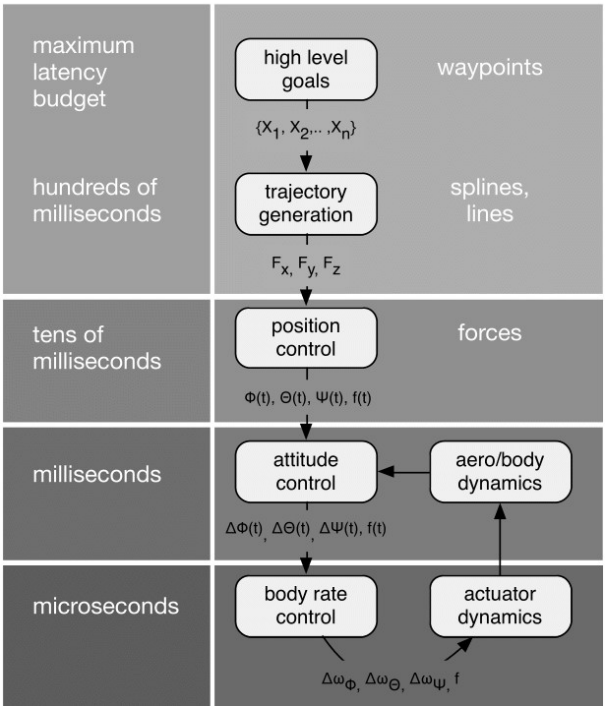

Figura 19 – Restrições de tempo em diferentes camadas para controle de um veículo. Adaptado de [39].

arquitetura. A camada mais baixa (camada de *hardware*) pode mudar bastante de acordo com o dispositivo, portanto, será apresentado somente a *Pixhawk*. Porém, deve-se ter em mente que o *firmware PX4* suporta inumeras plataformas de *hardware*.

3.3.1 *Drivers*

A camada acima do *hardware* é a camada dos *drivers* da *PX4*. Os *drivers* são o *software* básico que acessam e gerenciam todos os dispositivos do *hardware*, isto é, todos os sensores, motores, memórias e barramentos.

Os *drivers* disponibilizam interfaces para o sistema operacional, e este pode acessar os recursos do *hardware* sem necessitar saber todos os detalhes do mesmo. Cada plataforma suportada pela *PX4* como *Pixhawk*, *Intel Aero*, *MindPX*, entre outras, devem possuir *drivers*.

Os *drivers* da *PX4* fornecem um *timer* de alta resolução para medida de tempo e também permite que funções de tratamento sejam usadas juntamente com interrupções, o que é ideal para leitura de dados de sensores. Essa funcionalidade permite que a latência e *jitter* do sistema estejam abaixo de 4 microssegundos [39].

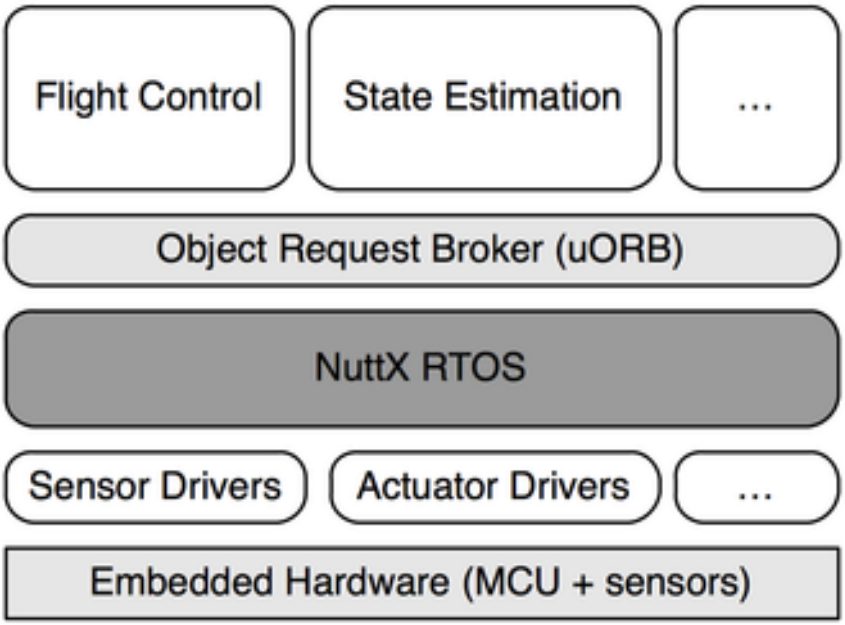

Figura 20 – Arquitetura da *PX4*. Adaptado de [39]

3.3.2 *NuttX RTOS*

Acima da camada de drivers está o sistema operacional *NuttX*, um sistema operacional de tempo-real que possui diversos recursos para sistemas embarcados, e possui ainda suporte para diversas *APIs* do *UNIX* (governado pelo padrão *POSIX*), o que facilita o desenvolvimento para a plataforma.

O *NuttX* é um sistema operacional completamente preemptível e suporta herança de prioridades, o que é ideal para as aplicações de tempo-real. O sistema operacional fornece bibliotecas C/C++ além de suporte para diversas arquiteturas de processadores como o *ARM Cortex* utilizado pelo *hardware Pixhawk* [25].

A versão do *NuttX RTOS* que a *PX4* utiliza é uma versão modificada para o projeto. Testes realizados mostraram que a troca de contexto nesse sistema operacional executando no *hardware* da *Pixhawk* dura cerca de 25 microssegundos, um valor bem baixo e que promove alta perfomance de *software* [39].

3.3.3 *uORB (PX4 middleware)*

A camada acima do sistema operacional, *Micro Object Request Broker (uORB)*, é a camada do *PX4 middleware*. *Object Request Broker* ou agente de requisição de objeto é um *middleware* que segue um padrão de projeto de *software* conhecido como *publish-subscriber*, que funciona conforme apresentado na Figura 21. Uma aplicação pode se inscrever (*subscribe*) em um tópico de interesse, e consequentemente ler os dados que estão sendo publicados nesse tópico, ou publicar(*publish*) dados nesse tópico.

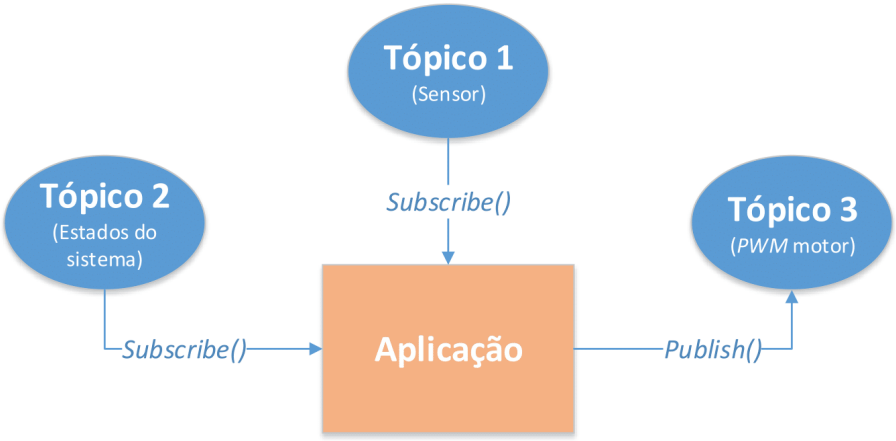

Figura 21 – Método *publisher-subscriber*

Um tópico não é nada mais que um canal de mensagens que permite que várias aplicações possam requisitar ou publicar dados. *Sensor girômetro* pode ser um nome de um tópico por exemplo. Nesse caso, um *driver* pode se inscrever para publicar nesse tópico os valores atuais do sensor girômetro, e uma aplicação de controle de um veículo poderia se inscrever nesse tópico e aguardar para receber dados a cada 50 ms, por exemplo. O sistema *uORB* foi desenvolvido de tal maneira que permite concorrência eficiente entre processos (*threads*[3] de tempo-real), podendo assim existir múltiplas aplicações inscritas nos mesmos tópicos. Essa arquitetura promove uma abstração no acesso dos dados dos dispositivos, sensores e atuadores, o que facilita o desenvolvimento de *software* para a plataforma.

3.3.4 *PX4 Flight Stack*

A camada do topo, também conhecida como *PX4 flight stack*, é a camada de aplicação em alto nível do sistema. Possui as chamadas aplicações *standalone*: controladores de vôo, estimadores de estados, entre outras, que podem ser escritas tanto em linguagem C quanto em C++. Essas aplicações são executadas e gerenciadas pelo sistema operacional, que irá executá-las conforme sua prioridade.

Na Figura 22 pode-se verificar a arquitetura da *PX4 flight stack*. As setas na cor azul representam a comunicação entre os módulos através do método *uORB*. Os módulos dentro dos retângulos pretos são módulos que não dependem de um veículo específico, e portanto, funcionam para qualquer plataforma. Já os módulos envolvidos pelo retângulo laranja são módulos específicos.

Pode-se ver abaixo, uma lista dos principais módulos de *software* até o momento

[3] É apresentado uma introdução sobre *threads* nos Apêndices.

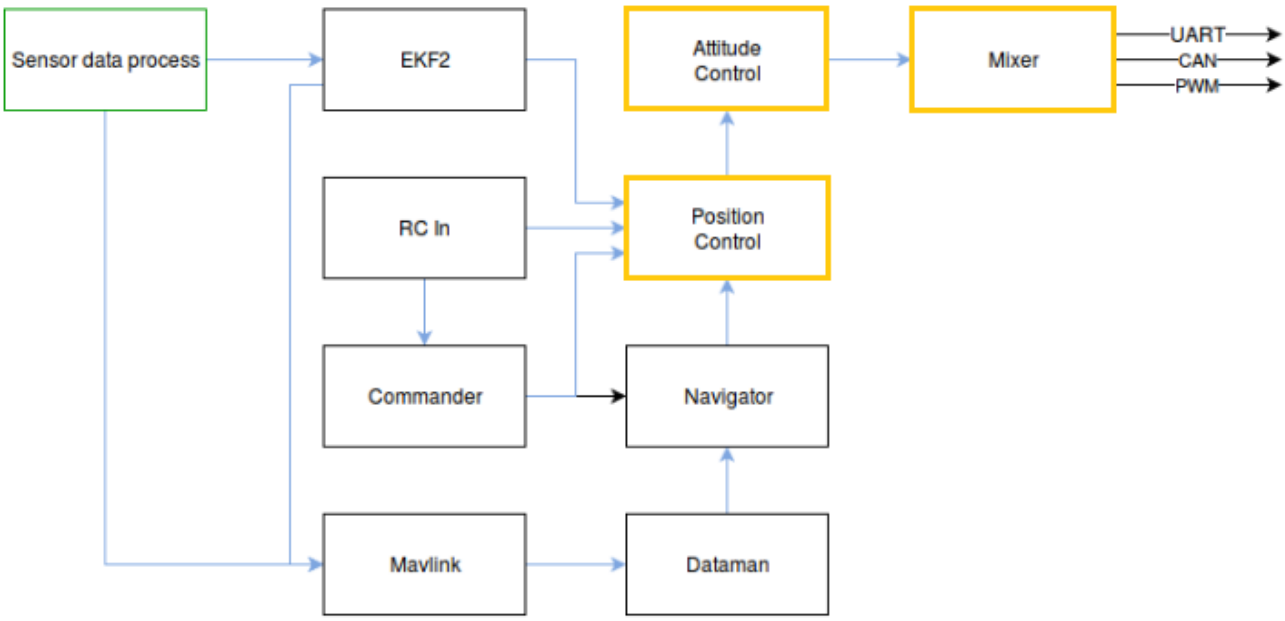

Figura 22 – Principais módulos da arquitetura de *software* de alto-nível da *PX4*. Adaptado de [10]

em que este trabalho foi desenvolvido:

- *Sensor data process*
- *RC In*
- *Mavlink*
- *Dataman*
- *Commander*
- *Navigator*
- *EKF2*
- *Position Control*
- *Attitude Control*
- *Mixer*

3.3.4.1 *Sensor data process*

Esse é o módulo de aquisição e ajuste dos dados dos sensores. Este acessa os dados dos sensores através dos *drivers* e mapeia em tópicos onde as aplicações poderão ter acesso pelo método *uORB*.

Esse módulo também faz ajustes e processamento em cima dos dados dos sensores, como por exemplo, compensação de temperatura para os acelerômetros, girômetros e barômetros. A calibração dos sensores também é gerenciada por este módulo.

3.3.4.2 RC In

Possui como objetivo adquirir os dados advindos do transmissor de rádio frequência, que é geralmente, o controle *RC* operado pelo usuário.

Esse módulo faz aquisições de tempo-real com uma frequência de 50 Hz [39]. Os dados são então decodificados através do hardware e dos drivers do receptor *RC*, e em seguida publicados em tópicos, de maneira a disponibilizar os comandos do transmissor *RC* para as aplicações seguindo o padrão de projeto *uORB*.

3.3.4.3 *Mavlink*

Esse módulo centraliza a comunicação do *hardware* da *PX4* com os demais dispositivos utilizando-se do protocolo *MAVLink*. Esse protocolo foi inventado em 2009 para comunicação com pequenos veículos não tripulados, e sua função é codificar mensagens e transmiti-las através de canais de comunicação serial [35].

O protocolo *MAVLink* é utilizado em larga escala na arquitetura da *PX4*. De maneira geral, o módulo *Mavlink* codifica todos os dados publicados nos tópicos da *PX4* (seja sensores, atuadores ou estados do sistema) em mensagens *MAVLink*. Desta maneira, para que a *PX4* se comunique com qualquer outro sistema, basta que este possua um decodificador de mensagens *MAVLink*. Os *softwares* de navegação[4] e os simuladores são exemplos de aplicação que comunicam com a *PX4* através de mensagens *MAVLink*. Computadores *offboard* como *Raspberry PI* são exemplos de computadores externos que se comunicam com a PX4 através de mensagens *MAVLink*.

A estrutura de um pacote *MAVLink* pode ser visto na Figura 23. O pacote possui entre 8 e 263 *bytes*. O *byte ID* da mensagem define como a mensagem deve ser decodificada. Em [8] pode-se encontrar diversas mensagens *MAVLink* que a *PX4* reconhece. O módulo *Mavlink* é encarregado de codificar e decodificar essas mensagens.

3.3.4.4 *Dataman*

Módulo que não possui documentação, além do código-fonte em [52]. Lida com o armazenamento de dados de missões e *frame* do veículo (quadrirotor tipo *X* ou +, por exemplo), podendo gravar estes dados tanto no cartão *SD* do *hardware* que utilizará a *PX4* ou na memória interna.

[4] *Qgroundcontrol* e *Mission Planner* são dois softwares de navegação que suportam o protocolo *MAVLink*.

Byte	Conteúdo	Explicação
0	Início do pacote	Indica o início do pacote
1	Tamanho da mensagem	Indica o tamanho da mensagem contida em payload
2	Sequência do pacote	Sequência usada para detecção de perda de pacotes
3	ID do sistema	Identificação do sistema, usado para diferenciar diferentes veículos na mesma rede
4	ID do componente	Identificação do componente, usado para diferenciar diferentes componentes de um mesmo sistema
5	ID da mensagem	Identificação da mensagem, define o que é a mensagem que está sendo enviada
6 à n+6	Payload	A mensagem a ser transmitida
n+7 à n+8	Checksum	Usado para deteção de erros

Figura 23 – Estrutura de uma mensagem *MAVLink*. O valor de n vai depender do tamanho da mensagem, sendo que esta deve respeitar o tamanho de 8 à 263 *bytes*.

3.3.4.5 *Commander*

Esse módulo contêm as aplicações encarregadas de controlar a máquina de estados principal do sistema. Também controla os *LEDs* do *hardware* que indicam o estado atual.

O *commander* é responsável pela "lógica de negócios"do sistema, garantindo que as demais aplicações executem ações que façam sentido no contexto atual do sistema. Um exemplo disso é: o *commander* não irá deixar o veículo iniciar o modo de vôo automático se o *GPS* não estiver conectado. As transições entre os modos de vôo também são gerenciadas por esse módulo, tornando este, um componente crítico do sistema.

3.3.4.6 *Navigator*

Módulo responsável por controlar a navegação do veículo. É um módulo bem extenso e que não possui documentação. As informações descritas nesse tópico foram todas obtidas através de análise do código-fonte em [52].

Entre as suas principais funções está a geração das trajetórias de baixo nível para o controlador de posição. Essa funcionalidade é de fundamental importância para o vôo automático do quadrirotor. O *software* de missão gera *waypoints*, ou pontos de navegação, para o veículo alcançar. O *navigator* particiona esses pontos em trajetórias menores e envia estas ao controlador de posição. Uma vez que o controlador de posição alcançou a posição desejada, este avisa para o *navigator* e então uma nova posição é gerada.

3.3.4.7 *EKF2*

Este módulo é relacionado a implementação do estimador de estados da *PX4 flight stack*, que usa um filtro de *Kalman* extendido. O filtro utiliza-se dos seguintes dados em sua entrada:

- Sensores da *IMU*: fornecem a variação angular (girômetro) e a variação da velocidade (acelerômetro) .
- Magnetômetro: fornecem dados de orientação magnética
- Altura: requer pelo menos uma fonte de dados de altura quer seja de um barômetro, *GPS* ou de um sensor externo.
- *GPS*: fornece dados que serão usados para posição e velocidade
- Velocidade do vento: velocidade do vento que serão usados para diminuir oscilações do veículo

Além desses items, o estimador ainda se utiliza de sensores de visão computacional, fluxo óptico, sensor *rangefinder*, entre outros[5] para o cálculo dos estados, uma vez que estes sensores estejam disponíveis no *hardware* utilizado.

Apesar da sua grande carga computacional, o *EKF* possui a vantagem de ser capaz de efetuar uma fusão de um considerável número de sensores a fim de estimar uma grande quantidade de estados com uma ótima performance [5].

A matriz de estados estimados disponibizada pelo *EKF* é uma matriz com 32 posições, apesar de que esta só possui 23 estados até o momento em que esse trabalho foi escrito. O *EKF* estima também as covariâncias para cada um desses estados. São eles:

- **[0-3]** Quatérnios, ou seja, a rotação em que o corpo está sujeito no exato momento.
- **[4-6]** Velocidade do corpo em relação à um frame fixo em terra.
- **[7-9]** Posição do corpo em relação à um frame fixo na terra.
- **[10-12]** Erros de medição (*bias*) dos ângulos da *IMU*.
- **[13-15]** Erros de medição (*bias*) de velocidade da *IMU*
- **[16-18]** Campo magnético terrestre
- **[19-21]** Campo magnético do corpo

[5] O código-fonte deste módulo é constantemente modificado e não possui documentação atualizada, portanto é difícil saber todos os sensores que este utiliza.

- **[22-23]** Velocidade do vento (norte-leste)

Esses dados são passados para outros módulos através do padrão *uORB*. O *EKF* publica os dados de saída em diferentes tópicos (de acordo com o tipo de dado), onde aplicações poderão se inscrever.

Os estados mais importantes, que refere-se à *IMU*, como os quatérnios e as velocidades, são amostrados à uma taxa de 250 Hz de acordo com testes realizados, até o momento em que este trabalho foi escrito.

3.3.4.8 *Position Control*

Esse módulo é responsável pelo controlador de posição do veículo. Cada tipo de veículo deve possuir seu próprio módulo para controle de posição.

Para o quadrirotor, esse módulo recebe as referências de posição (Xsp,Ysp,Zsp), ou seja, para onde o veículo deverá se locomover, conforme Figura 24. À partir daí, é calculado os valores de velocidade linear em que o quadrirotor deve estar, tendo-se como base os dados da posição atual advindos do estimador de posição (*EKF*). Para o cálculo das velocidades lineares, utiliza-se de um pré-compensador para cada uma das entradas (X,Y,Z), bloco P da Figura 24.

A saída do bloco do pré-compensador alimenta o bloco de controle *PID*, onde são gerados as referências de ângulo *roll*, *pitch* e *yaw*. É gerado também o valor desejado de empuxo (*throttle*) que deve ser aplicado no veículo para este ganhar altura. Para chegar a esses valores de saída, o controle *PID* é aplicado em cima de cada uma das velocidades lineares, tendo como base os estados atuais vindos do estimador de posição.

A descrição deste módulo foi apresentada de maneira simplificada. Este módulo possui importante funções de acordo com o modo de vôo que está sendo selecionado. Se um modo manual for escolhido por exemplo, os estados desejados serão mapeados diretamente do controle de rádio *RC* e não mais dos controladores apresentados.

3.3.4.9 Attitude Control

Módulo responsável pelo controle em baixo nível das atitudes ou ângulos do veículo. Esse módulo é dependente do tipo de veículo.

Para um quadrirotor, esse módulo é responsável por implementar o controle dos ângulos *roll*, *pitch* e *yaw*, e consequentemente, mover e estabilizar o veículo no ar. Este módulo recebe os estados atuais e o estados desejados (*set points*), e à partir daí, implementa dois controladores (Figura 25).

O primeiro é um controlador proporcional (controlador P) responsável pelo controle dos ângulos. Esse controlador recebe os estados atuais em termos de *roll*, *pitch* e

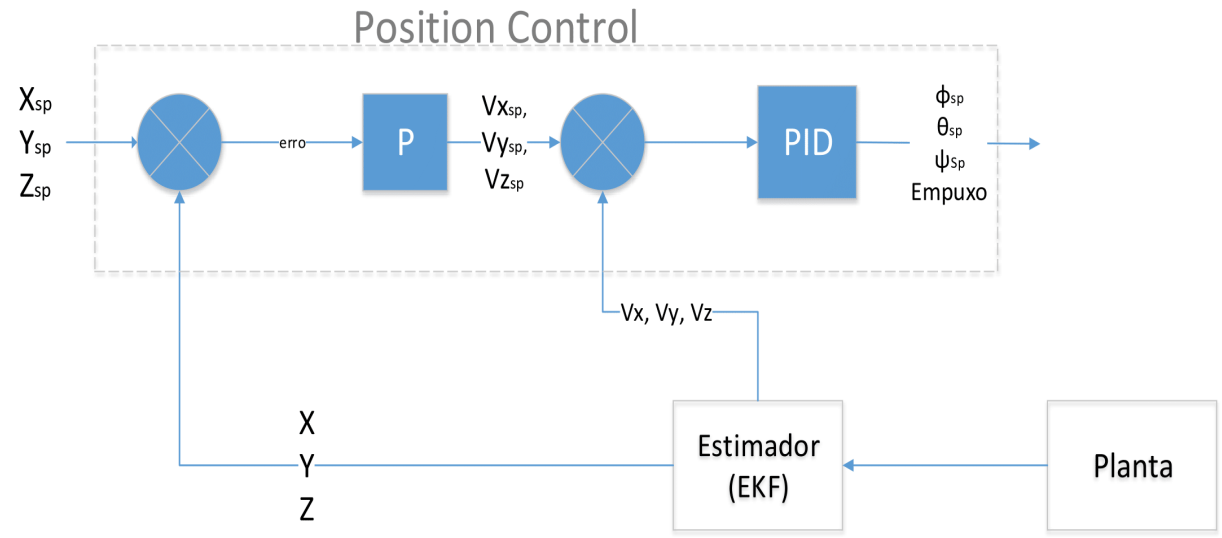

Figura 24 – Arquitutetura do módulo *Position Control*

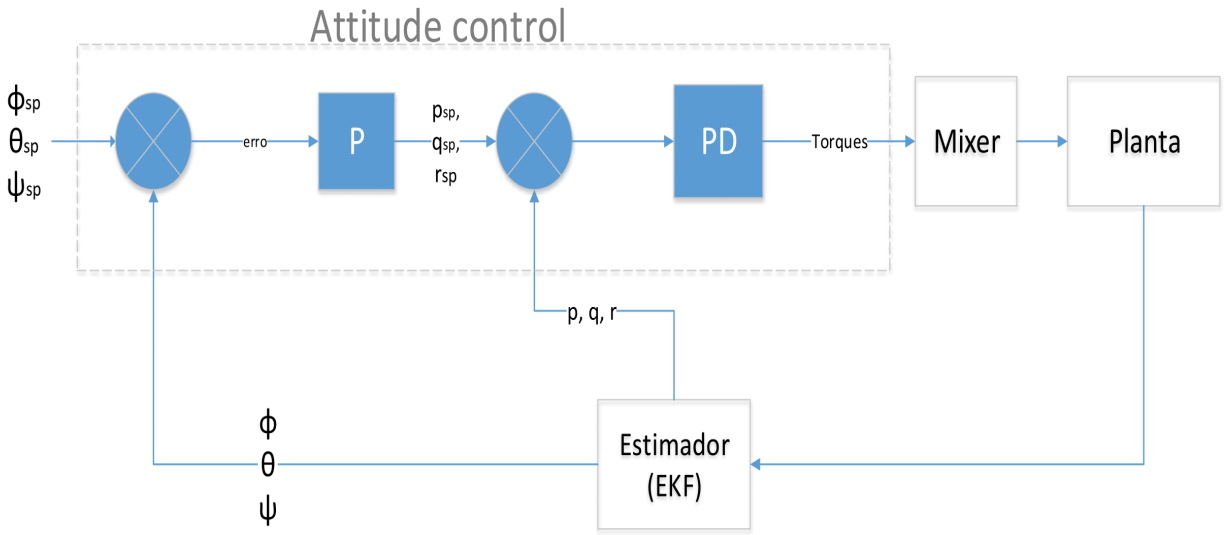

Figura 25 – Arquitutetura do módulo *Attitude Control*

Figura 26 – Entradas e saídas do módulo *Mixer*

yaw vindo do estimador de atitude (EKF) e recebe também os *set points*. A saída deste controlador são as velocidades angulares desejadas.

O segundo controlador recebe as velocidades angulares desejadas, vindas do primeiro controlador, e recebe também as velocidades estimadas vindas do estimador de atitude. A técnica *PD* (proporcional derivativo) é então aplicada nestes valores, e a saída é os torques para os motores.

O controle de atitude deve ser executado com alta prioridade no sistema operacional, pois este precisa obter os estados dos sensores e calcular comandos para os atuadores com uma frequência ideal maior ou igual à 20 Hz[6], ou período menor que 50 ms. Esse tempo deve ser garantido, portanto a alta prioridade de execução pelo escalonador é obrigatória.

3.3.4.10 *Mixer*

Esse módulo é responsável por "mixar"os comandos de torque recebidos dos módulos de controle, e à partir daí, gerar sinais de controle para os motores (Figura 26).

O *Mixer* é implementado para cada tipo de veículo. Para o quadrirotor, o *Mixer* baseia-se no *frame* do veículo utilizado, e à partir daí, é possível calcular o peso que o torque em cada dinâmica irá exercer. Para ilustrar isto, basta lembrar que um quadrirotor pode não ter os braços simétricos, e que portanto, torques em *pitch* e *roll* terão pesos diferentes.

Como pode se ver, a vantagem desse módulo é que, ao separá-lo da lógica de controle, a reusabilidade de código é melhorada [13]. O controlador não precisa saber o tipo *frame* do veículo que será utilizado.

O *Mixer* também executa ajustes para manter um vôo mais robusto. Conforme pode-se ver no código fonte [52], o algoritmo utilizado dá uma grande importância para os torques de *roll* e *pitch*, e dá menor importância para o torque de *yaw* na geração das

[6] Valor estipulado, de acordo com a menor taxa de amostragem dos sensores. Não foram encontrados estudos que estipulassem um valor mínimo.

saídas dos motores. Acredita-se que a razão disso, apesar de não existir documentação que comprove, é o fato dos movimentos de *yaw* não exercerem uma influência significativa na estabilização do veículo.

Para o quadrirotor, os torques calculados pelo controlador para cada ângulo de Euler (τ_ϕ, τ_θ, τ_ψ) são normalizados entre -1 e 1. Já o movimento de *throttle* é normalizado entre 0 e 1. Essas forças são então "*mixadas*"para gerar saídas para cada um dos quatro motores. As saídas geradas para cada motor estão entre -1 e 1. Esses valores serão utilizados pelo *driver PWM* para calcular os comandos para os *ESCs*.

3.3.5 *Hardware*

Diversas plataformas de *hardware* são capazes de executar o *firmware* da *PX4*. Algumas delas são: *Pixhawk*, *Intel Aero*, *Parrot Bebop 2*, *MindRacer* e *Pixfalcon*.

O hardware escolhido para este trabalho foi a *flight controller* a *Pixhawk*. Ela foi desenvolvida para suportar especialmente a *PX4 flight stack*, apesar de suportar também a *Arduino Pilot Meta (APM)*.

A *Pixhawk* é uma plataforma de *hardware* aberto, ou seja, todos os detalhes deste produto está disponível e pode ser replicado. Portanto, existem várias versões desse produto, sendo que para este trabalho utilizou-se da versão da *3DR Robotics*[7]. As principais características são:

- Microcontrolador embarcado: STM32F427 Cortex M4 32-bit
  - *clock* de 168 MHz
  - *FPU* (*float-point unit*) [8]
  - 2MB de memória *Flash*
  - 256 KB de memória RAM
- Microcontrolador para segurança contra falhas: STM32F103 Cortex M3 32-bit
- Sensores de movimento
  - Sensor integrado giroscópio/acelerômetro de 3 eixos
  - Giroscópio adicional com resolução de 16 bits
  - Sensor integrado acelerômetro/magnetômetro com resolução de 14 bits
  - Barômetro
- Interfaces

[7] *www.3dr.com*
[8] Proporciona aumento de perfomance em operações com ponto flutuante;

- 4 portas *UART*
- 2 portas *CAN*
- 1 porta *I2C*
- 1 porta *SPI*
- Conversores A/D de 3.3V e de 6.6V
- Entrada para espectro *DSM* [9]
- *S.BUS*[10]
- Entrada para sinal PPM
- 8 portas de entrada/saída para sinal *PWM*
- *MicroUSB* para comunicação com um computador

O esquemático completo do projeto da *Pixhawk* pode ser encontrado em [12].

3.4 Simulação

A arquitetura da *PX4* possibilita a execução de simulação para testes do *software* antes de embarcá-lo em um veículo real. Esses testes são necessários para verificar, principalmente, se a aplicação de controle do veículo está funcionando de acordo com o esperado. A *PX4* fornece dois contextos de simulação: *Hardware-in-the-loop* (*HIL*) e *Software-in-the-loop* (*SIL*).

3.5 *Software-in-the-loop* (*SIL*)

De maneira à avaliar a performance da aplicação de controle antes de executá-la no *hardware target* (seja um sistema embarcado ou qualquer tipo de sistema final que irá executar a aplicação), recomenda-se utilizar *SIL*.

Este tipo de simulação funciona da seguinte maneira: um computador com muitos recursos disponíveis (*PC* ou *notebook*) irá executar a aplicação de controle dentro de um ambiente de *software* modelado para um quadrirotor. Uma vantagem é a identificação de *bugs* com ferramentas de *debug* avançadas e grande disponibilidade de recursos[11].

A *PX4* fornece dois ambientes de simulação *SIL*, ambos com interface em *3D*. Um deles, é o *jMAVSim*, simulador gráfico simples desenvolvido na linguagem *Java* por um dos desenvolvedores da *PX4* cujo código-fonte pode ser encontrado em [18]. O outro

[9] Usado para conectar dispositivo de telemetria
[10] Barramento para conectar múltiplos motores
[11] Computador moderno comum possui processadores de oito núcleos e vários *gigabytes* de memória *RAM*.

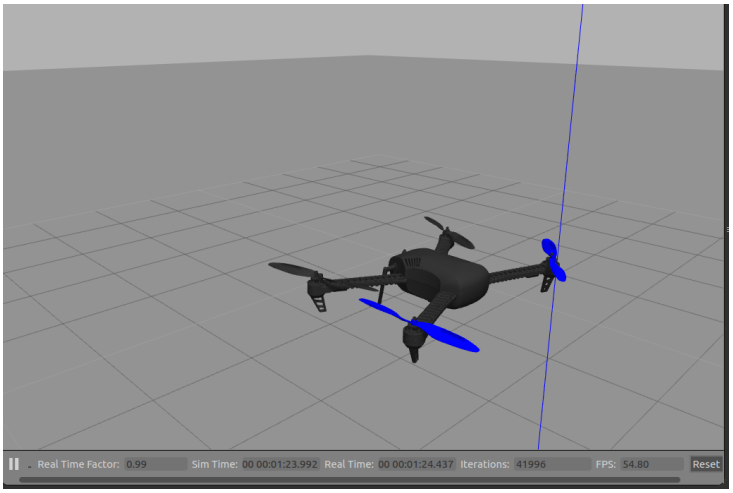

Figura 27 – Simulador *Gazebo*

simulador bastante usado é o *Gazebo*. Este simulador, cujo desenvolvimento começou em 2002, é bem conhecido no mundo da robótica, pois possibilita simulações mais complexas e possui diversos desenvolvedores ao redor do mundo. A equipe de desenvolvedores da *PX4* modelou alguns veículos para este simulador, e portanto, é possível utilizá-lo para testar as estratégias de controle. Podemos ver na Figura 27 o quadrirotor *Iris* da empresa *DJI* no *Gazebo*.

3.6 *Hardware-in-the-loop* (*HIL*)

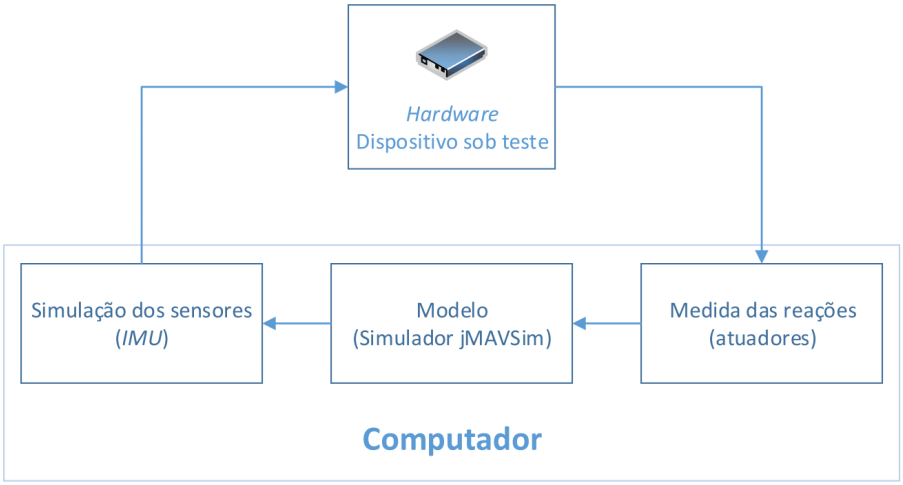

Figura 28 – *Hardware-in-the-loop* (*HIL*)

Este tipo de simulação ocorre utilizando um software de simulação em *3D*, porém, ao invés de executar a aplicação no computador do usuário, a aplicação executa no

hardware que será utilizado em vôo. Os sensores e atuadores são simulados através do *software* de acordo com a resposta vinda do *hardware* embarcado, que se comunica com o computador, geralmente, através da porta serial.

Este tipo de simulação possibilita identificar como a aplicação está se comportando no sistema embarcado, e é, mais uma maneira de validar a estratégia de controle, uma vez que é possível observar o tempo de resposta e os comandos gerados pelo controlador. Até o momento em que este trabalho foi escrito, a simulação em *HIL* da PX4 para veículos do tipo quadrirotor funciona somente com o software *jMAVSim* citado na seção anterior.

4 Metodologia

Esta seção apresenta a metodologia que foi adotada para o desenvolvimento deste trabalho seguida do detalhamento de cada etapa do processo. O trabalho foi dividido em cinco etapas: revisão bibliográfica, familiarização com o ambiente de desenvolvimento, definição do problema estudos de caso e análise dos resultados[1]. A Figura 29, apresenta estas etapas em forma esquemática.

4.1 Etapa 1 - Revisão Bibliográfica

A primeira etapa foi uma das mais extensas do trabalho, pois foram revisados todos os conceitos importantes para a elaboração do projeto.

Foi necessário o estudo de funcionamento do quadrirotor incluindo modelo matemático, técnicas de controle, sensores e demais dispositivos para vôo. Uma extensa revisão de conceitos de sistemas embarcados foi realizada da mesma forma, para possibilitar o entendimento destes conceitos aplicados para o quadrirotor, uma vez que não é comum encontrar trabalhos em que relacione os dois.

4.2 Etapa 2 - Familiarização com o ambiente de desenvolvimento

Esta etapa foi sem dúvida a etapa mais longa do projeto, juntamente com a Etapa 1. Ela consistiu em estudar toda a arquitetura do *firmware PX4* e o *hardware Pixhawk*.

O estudo do *firmware PX4* foi elaborado tendo como base a documentação existente, que embora desatualizada serviu para início do estudo, e, principalmente, o código-fonte. De maneira a entender o funcionamento, módulos foram constantemente modificados para fins de teste. Uma vez que o entendimento das aplicações principais foram alcançados, uma revisão da arquitetura foi elaborada.

4.3 Etapa 3 - Definição do problema

Devido ao fato de o quadrirotor possuir seis graus de liberdade e apenas quatro entradas (rotação dos motores), este é considerado um sistema sub-atuado. Os movimentos de translação e rotação são acoplados à fim de atingir a completa movimentação nos seis graus de liberdade, e como consequência, a sua dinâmica é extremamente não-linear o que torna o controle deste veículo um problema interessante na teoria de controle [16].

[1] Estas etapas não refletem a estrutura de capítulos do trabalho.

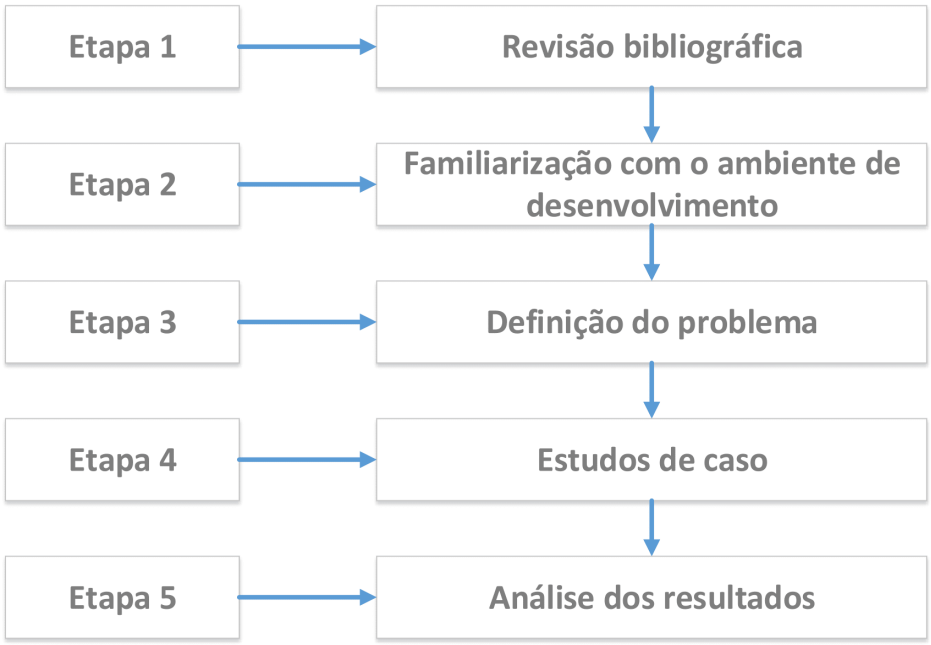

Figura 29 – Etapas do projeto

Várias estratégias como *PID* e *LQR* mostraram-se capazes de resolver o problema de controle do quadrirotor, porém, a implementação destas técnicas necessitam de um conhecimento de sistemas computacionais e eletrônicos, para integrar múltiplas aplicações de *software* com diferentes restrições de tempo em um *hardware* com um sistema operacional de tempo-real embarcado, e garantir que a aplicação de controle do veículo tenha latência e *jitter* pequenos e bem delimitados.

O *firmware PX4* une *drivers* para diversas controladoras de vôo, um sistema operacional de tempo-real (*NuttX*) e um *middleware* para comunicação entre módulos criando uma plataforma eficiente para aplicações de controle de vôo, porém, desenvolver para esta plataforma é um desafio pelo fato de não possuir uma documentação adequada. Além disso, ao promover um sistema que prioriza reusabilidade de código e suporte multiplataforma, a *PX4* utiliza muito mais do processador do que necessário para uma aplicação específica.

Estes problemas podem ser solucionados ao retirar o módulo de controle oficial da *PX4* e implementar estratégias próprias de controle de maneira computacionalmente

otimizada, à fim de validar a teoria por trás destas técnicas e mostrar que existem outras alternativas aos usuários da plataforma *PX4* que não desejam o utilizar o controlador padrão.

4.4 Etapa 4 - Estudos de caso

Os estudos de caso apresentaram as implementações das técnicas de controle *PID* e *LQR*. Para cada estudo de caso adotou-se a seguinte estratégia: implementação em *software-in-the-loop* (*SIL*), implementação em *hardware-in-the-loop* (*HIL*), testes em bancada e testes em vôo, sendo que estas etapas eram seguidas sequencialmente, e somente avançava para a etapa seguinte quando resultados satisfatórios eram alcançados.

4.5 Etapa 5 - Análise dos Resultados

Por fim, na etapa cinco, foi executada uma análise e discussão dos resultados obtidos. Essa etapa conclui o trabalho proposto, realizando uma análise dos resultados obtidos e os resultados esperados, além de uma comparação entre os modelos implementados, e os modelos que já são previamente implementados ao adquirir-se uma das placas de controle já existente no mercado. Uma análise do impacto dessas técnicas implementadas para a literatura é também apresentada.

5 Estudo de Caso

5.1 Estudo de caso

Esta seção apresenta a descrição dos estudos de caso realizados. Dois estudos foram desenvolvidos, sendo que o primeiro trata-se da implementação de um controlador *PID* e o segundo trata-se da implementação de um controlador *LQR* ambos embarcados na plataforma de vôo *PX4 Autopilot* (*firmware PX4* e *Pixhawk*), à fim de reutilizar módulos que demandam muito tempo para implementação (*drivers* por exemplo). O quadrirotor a ser utilizado foi montado no laboratório da *UnB* usando-se um *kit* do *frame F450* da empresa *DJI*.

5.2 Caso 1: Controlador *PID*

O primeiro estudo de caso deste trabalho é a implementação de um controlador *PID* para os ângulos de Euler (atitude) na controladora de vôo *Pixhawk*. Apesar dessa arquitetura já possuir um controlador bem similiar ao *PID* [47], a arquitetura proposta do *PID* à ser implementado será diferente.

Conforme visto na Figura 25, o controle de atitude padrão da *PX4* possui dois controladores em sequência (ou cascata). O primeiro gera velocidades angulares desejadas (p_d,q_d,r_d) a partir dos ângulos de Euler desejados (ϕ_d,θ_d,ψ_d) utilizando um controlador *P*. O segundo recebe os valores de velocidades angulares desejadas e calcula os torques para os atuadores em cada direção (τ_ϕ,τ_θ,τ_ψ).

A topologia do controlador *PID* proposta neste trabalho de conclusão de curso[1] consiste em apenas um controlador *PID* para os ângulos de Euler desejados (ϕ_d,θ_d,ψ_d) e a saída são os torques em suas respectivas direções (τ_ϕ,τ_θ,τ_ψ).

Para implementar esse controlador foi adequada a seguinte estratégia: retirar o módulo *control attitude* conforme a Figura 22, e implementar um módulo próprio para executar o controle.

A vantagem da técnica adotada está no fato de conseguir compreender todos os detalhes da implementação em baixo nível do controlador. A desvantagem está no fato de que, ao substituir todo o módulo, diversos detalhes devem ser tomados em consideração, principalmente em relação à modos específicos de vôo.

Vale ressaltar que todos os passos descritos neste trabalho foram realizados utili-

[1] Disponiblizada pelo aluno de graduação Henrique Medeiros.

zando a versão de *firmware v1.5.5* [11].

5.2.1 Entradas e Saídas

O módulo de controle que foi desenvolvido, troca informações principalmente com o módulo de controle de posição e o *mixer*, conforme o módulo de controle de atitude oficial da *PX4* visto na Figura 22.

Para implementar a aplicação, foi efetuado primeiramente um estudo sobre os tópicos de interesse. Tópicos, conforme anteriormente vistos, são canais de comunicação que permite que múltiplas aplicações se comuniquem entre si. O controlador implementado usa alguns tópicos como entradas e saídas de dados. Abaixo tem-se uma lista destes tópicos:

- *vehicle_attitude_setpoint*: fornece dados de *set point*. É utilizado para adquirir os ângulos de Euler desejados (ϕ_d,θ_d,ψ_d)
- *vehicle_attitude*: fornece dados sobre o estado atual do veículo, vindo do estimador. É utilizado para adquirir o estado em que o veículo se encontra em termos de (ϕ,θ,ψ)
- *vehicle_control_mode*: fornece dados sobre os modos de controle do veículo. É utilizado para sinalizar que o controle deve ser efetuado caso algumas condições estejam habilitadas
- *actuator_armed*: informa se os motores do veículo estão armados. É utilizado para verificar se o controle deve ser executado
- *multirotor_motor_limits*: informa se os motores do veículo atingiram a saturação. É utilizado para cancelar os termos integrais de controle caso os motores estejam enfrentando saturação
- *parameter_update*: informação sobre a atualização de parâmetros. É utilizada para adquirir importantes parâmetros em tempo real, como mudança nos ganhos do controle
- *actuator_control*: tópico com dados sobre os torques dos atuadores. É utilizado para publicar os valores de torques.

Destes tópicos de interesse, somente o último é utilizado para executar as ações de controle. Os demais tópicos são dados de entrada para a aplicação implementada.

5.2.2 Aplicação

O código de controle foi implementado em C e C++ utilizando o mesmo padrão dos outros códigos escritos para a *PX4*. Como a biblioteca de C++ do *firmware* é um pouco precária[2], algumas bibliotecas auxiliares foram escritas em C. O algoritmo implementado possui uma classe principal que encapsula todos os métodos e dados do controlador.

A arquitetura de *software* de controle pode ser visualizada na Figura 30. A aplicação é inicializada e em seguida executa um *loop* até que outra aplicação sinalize informando que não há mais necessidade de um controle de atitude, o que dificilmente ocorre, já que o veículo sempre irá necessitar desse controle.

Depois da rotina de inicialização onde se inscreve nos tópicos e inicializa as variáveis e estruturas, a aplicação entra no seu *loop* e a primeira coisa a se fazer é uma chamada à função *poll()* para inspecionar o tópico referente aos sensores da *IMU*. Esta é uma função dos sistemas *UNIX* cujo objetivo é inspecionar um descritor de arquivos (neste caso o tópico dos sensores *IMU*).

Ao realizar esta chamada, o sistema operacional bloqueia a execução da aplicação (*thread* em estado bloqueado), e só retorna para ela quando novos dados chegarem no arquivo inspecionado, ou caso o tempo se esgote. Isto é uma grande vantagem pois a *CPU* não gasta ciclos esperando que novos dados chegue, e enquanto isso, o sistema operacional pode executar outras tarefas. Um limite de 50 milissegundos[3] foi definido, ou seja, sempre quando o *loop* de controle inicia, a aplicação espera até 50 ms por alguma variação nos dados dos sensores da *IMU*. Caso não ocorra nenhuma alteração dentro deste tempo, ou caso ocorra antes, a aplicação retoma sua execução.

A aplicação segue então lendo os demais tópicos de entrada, conforme visto na seção anterior. Uma vez que todos os dados foram adquiridos, a aplicação irá checar com o módulo *commander* qual o modo de vôo em que se encontra o veículo e irá executar a função de controle *PID*.

A maneira em que o controle *PID* foi implementada será vista com detalhes na próxima seção. Dependendo do modo de vôo do veículo o controle *PID* para *yaw* será executado com base em ângulo desejado ou velocidade angular desejada, pois de acordo com [46], todos os modos de vôo manuais controlam *yaw* através da sua velocidade angular. Para todos os modos de vôo, o controle de *roll* e *pitch* é efetuado através dos ângulos, portanto o controlador não funciona para os modos de vôo *ACRO* e *RATTITUDE PX4*[4].

[2] Não possui a C++ *Standard Library* por exemplo.

[3] Valor empírico. Não foram encontrados trabalhos que definam um limite máximo de *loop* de controle para quadrirotor.

[4] Estes dois modos de vôo funcionam passando comandos de velocidades angulares desejadas. O controlador implementado possui como entrada ângulos e e como saídas torques, portanto não trabalha com velocidade angular para controle.

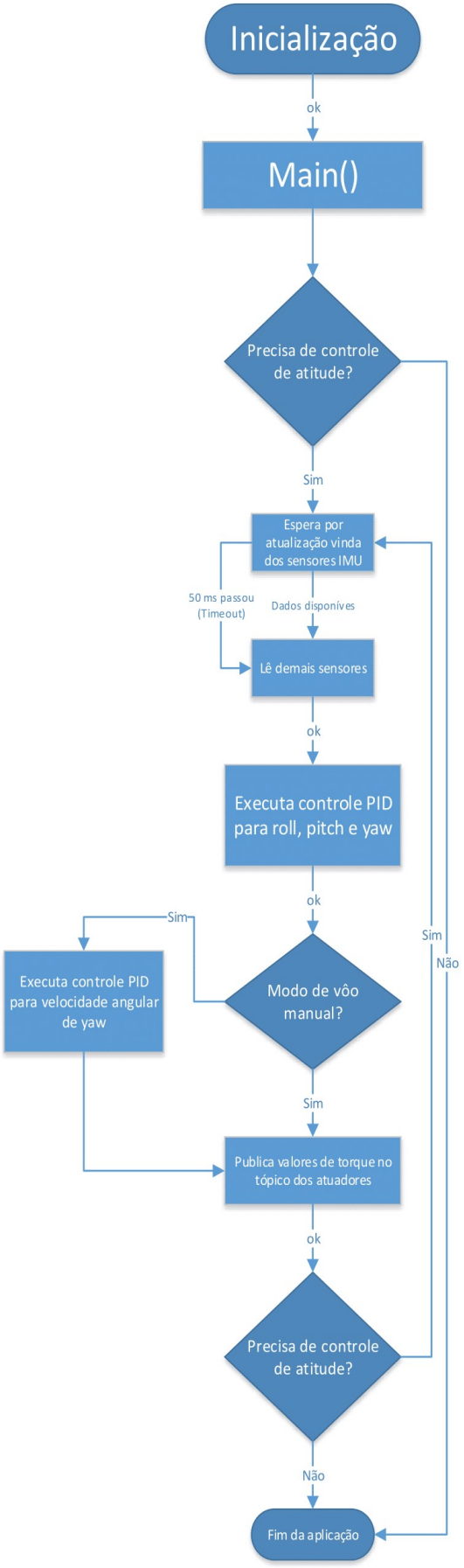

Figura 30 – Arquitetura da aplicação de controle *PID*

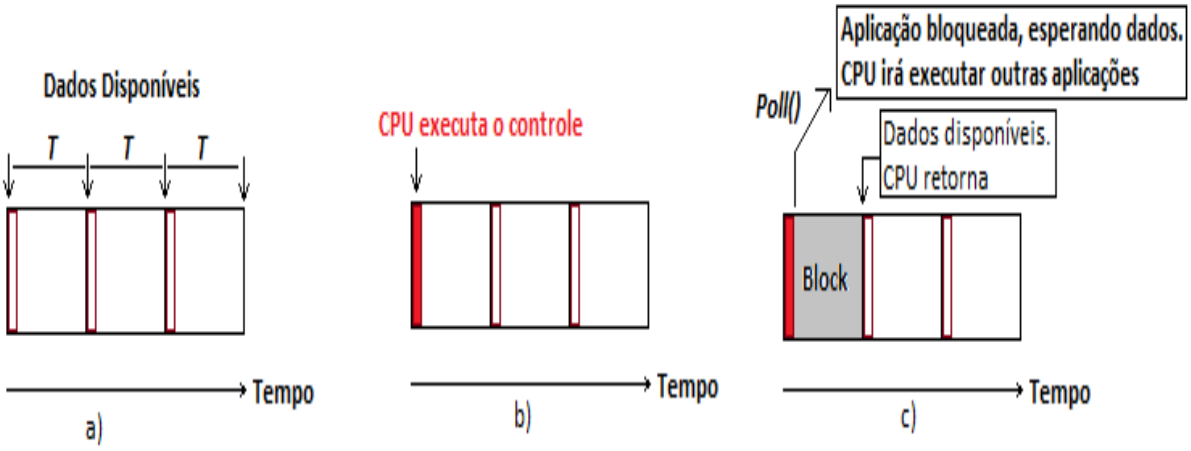

Figura 31 – a) Período de amostragem; b) CPU executa o controle PID; c) Troca de contexto

Por fim, a aplicação publica os valores de torque calculado para cada um dos atuadores no tópico *actuator_control*. Esse tópico é constantemente lido pelo módulo do *mixer* onde os torques são normalizados e transformados em saída para os motores.

5.2.3 Bloco *PID*

Para desenvolver o bloco de controle em *software*, tomou-se como base a equação discretizada do controlador *PID* [51]:

$$\begin{cases} u_i(k) = u_i(k-1) + K_{id}Te(k) \\ u(k) = K_{pd}e(k) + u_i(k) + \frac{K_{dd}}{T}(e(k) - e(k-1)) \end{cases} \tag{5.1}$$

em que $u(k)$ é o sinal de controle; $u_i(k)$ é a parte integral do controlador; K_{pd}, K_{id} e K_{dd} são os ganhos, e $e(k)$ é o sinal de erro. T é o período de amostragem e k é o número da amostra.

O período de amostragem T é medido através de uma chamada fornecida pelo sistema operacional da PX4, chamada *hrt_absolute_time()*. Essa função acessa o *timer* do microcontrolador, e fornece uma informaçao de tempo bem precisa.

Conforme visto no Capítulo 3, a arquitetura da *PX4* foi projetada de maneira a otimizar o tempo de latência. O período de amostragem será o tempo em que a aplicação executa o controle somado com o tempo que ela irá gastar esperando por novos dados do estimador de atitude chegar. Portanto, o controle deve ser executado em uma fração de tempo entre a chegada de dados para garantir que o sistema não sofra de latência (Figura 31). Sabendo-se que o módulo *EFK* disponibiliza dados em frequência de 250 Hz, o período de amostragem será de aproximadamente 4 ms, e deseja-se que o controlador seja executado em uma fatia deste tempo.

Período de amostragem (EKF)	4 ms
Tempo máximo de espera por novos dados antes de iniciar o controle (caso o EKF falhe em amostrar em 4 ms)	50 ms
Período de amostragem à ser utilizado pela aplicação de controle (medido à cada iteração)	1 ms <= T <= 10 ms

Figura 32 – Restrições temporais adotadas

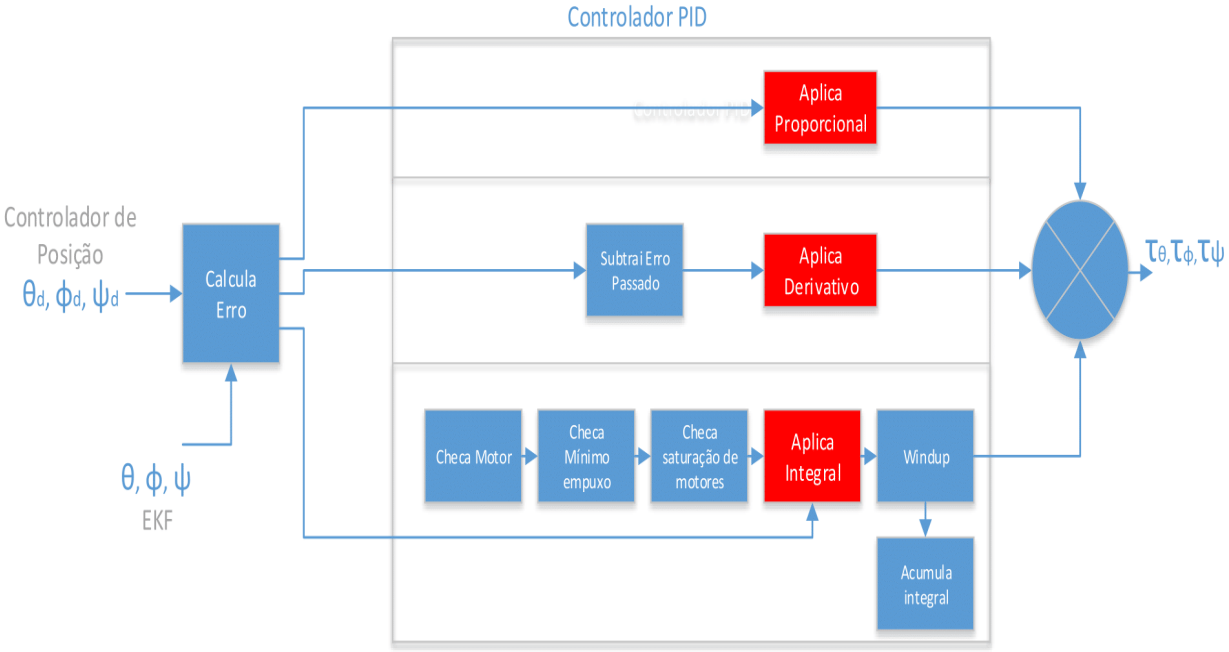

Figura 33 – Bloco de controle *PID* implementado em *software*

O período de amostragem, medido entre cada iteração, é resguardado à cada medição garantindo-se que não possa ter valores menores do que 1 ms ou maiores do que 10 ms, para evitar que ruídos (na leitura de tempo do *timer*, ou uma rara falha no módulo *EKF*) afete a aplicação de controle. A Figura 32 ilusta as restrições temporais adotadas neste trabalho.

A implementação pode ser vista na Figura 33. O erro é calculado subtraindo os valores desejados pelo valores atuais vindo dos sensores. Os ângulos atuais vêm do módulo *EKF* ao passo que os ângulos desejados vêm do controlador de posição para os modos de vôo automático. Para os modos de vôo manuais, os ângulos desejados (ou velocidades desejadas, no caso do *yaw*) são enviados através dos comandos do rádio.

O erro é primeiramente utilizado para calcular a parcela proporcional, onde é multiplicado pelo ganho proporcional, e o valor calculado é salvo. Em seguida subtrai-se o erro atual do erro passado, e aplica o ganho derivativo, novamente salvando-se o valor encontrado. O erro atual será salvo para o próximo laço de controle.

Por fim, a parcela do cálculo integral é a que demanda mais processamento devido ao fato de ser um valor acumulado com o tempo, portanto deve-se fazer alguns ajustes para evitar valores muito altos para o integral gerando oscilações.

Primeiramente checam-se os estados dos motores, pois caso estejam desarmados a parcela integral acumulada será zerada[5]. Em seguida aplica-se a mesma lógica caso o empuxo vertical seja muito baixo, o que significa que os motores estão armados mas o veículo está no chão. Em seguida, checa-se a saturação de cada um dos motores, caso estejam saturados, uma *flag* indica qual tipo de movimento saturou o motor (*roll* positivo, *roll* negativo, *pitch* positivo, etc), e aplica-se uma correção que não deixa veículo acumular a parcela integral nesse sentido, apenas permanece do jeito que está, ou, aceita se for um comando contrário à saturação. Por fim, aplica-se o ganho integral em cima do erro, e uma prevenção anti-*windup* é executada, que previne que valores acima do desejado para o integral sejam perpetuados, e por fim, a parte integral é acumulada para a próxima iteração.

Uma vez que todos os termos estejam calculados, soma-se as três parcelas, e os torques são publicados (via *uORB*) para cada um dos eixos.

5.3 Caso 2: Controlador *LQR*

O segundo estudo de caso deste trabalho é a implementação de um controlador Regulador Quadrático Linear (*LQR*) para controle de atitude do veículo quadrirotor embarcado no *hardware Pixhawk*.

O desafio inicial deste estudo de caso foi implementar uma técnica de controle que depende de um modelo do sistema a ser identificado, que neste caso é o quadrirotor disponível no laboratório. Outros alunos envolvidos no projeto conseguiram a identificação de alguns dos principais parâmetros do quadrirotor *F450*, e estes foram utilizados para a modelagem do controlador *LQR* (Tabela 1).

Uma vez com os principais parâmetros do modelo em mãos, uma linearização deste foi efetuada utilizando o *software MATLAB*. O algoritmo[6] lineariza o modelo em torno do ponto de equilíbrio escolhido (0,0,0,0,10,0,0,0,0,0,0,0) para cada variável de estado do quadrirotor (x,y,z,v_x,v_y,v_z,ϕ,θ,ψ,p,q,r). A matriz do sistema (A) e a matriz de entrada (B) são obtidas através deste algoritmo.

[5] Não há necessidade de acumular o termo integral se o veículo não está tentando estabilizar.
[6] Desenvolvido pelo Professor Renato Vilela

Ixx	0.019 $[kg \cdot m^2]$
Iyy	0.025 $[kg \cdot m^2]$
Izz	0.064 $[kg \cdot m^2]$
m	1.328 $[kg]$
g	9.782 $[m/s^2]$

Tabela 1 – Parâmetros identificados do sistema

Para calcular o ganho ótimo do *LQR* (K), foi utilizada a função *dqlr* do *MATLAB*. Um algoritmo que cria as matrizes de ponderação e calcula a matriz de ganho foi desenvolvido. Assim como no desenvolvimento do estudo de caso 1, por não ser o escopo deste trabalho, foram ignorados o controle das variáveis de estado (x,y,z,v_x,v_y,v_z) para este estudo de caso, utilizando-se então, o módulo *Position Control* da *PX4*.

5.3.1 Aplicação

As entradas e saídas do controlador são análogas àquelas apresentadas no estudo de caso 1, portanto não serão repetidas aqui.

A arquitetura da aplicação pode ser vista na Figura 34. Vários blocos são análogos ao estudo de caso 1. É utilizado a mesma técnica onde uma classe encapsula todas comportamentos e atributos, e para ser executada, o sistema operacional deve instânciar um objeto desta classe.

A aplicação é inicializada e logo em seguida uma função é chamada para ler o arquivo de texto gerado pelo *Matlab* que contêm a matriz dos ganhos. A matriz é armazenada em uma estrutura e a função principal *Main()* é chamada. Esta checa se o controle de atitude é necessário e caso a resposta seja positiva, é realizado um *polling* dos dados conforme explicado no estudo de caso anterior.

A diferença principal entre essa aplicação e a aplicação do controlador *PID*, está no fato de que esta executa uma rotina para montar os vetores de estado toda vez em que os dados dos sensores são atualizados. Uma vez montados os vetores, o bloco de controle *LQR*, cujo os detalhes serão discutidos na próxima seção, é executado.

O bloco de controle *LQR* produz valores de torque (τ_ϕ,τ_θ,τ_ψ) e, devido à estrutura do *Mixer* discutida anteriormente, os valores esperados estão na faixa de -1 e 1. Para o controlador funcionar corretamente, foi necessário uma rotina de mapeamento dos torques, que colocasse os valores calculados na faixa de trabalho do mixer.

Por fim, os torques mapeados são publicados no tópico dos atuadores, e a aplicação executa enquanto houver necessidade de controle de atitude.

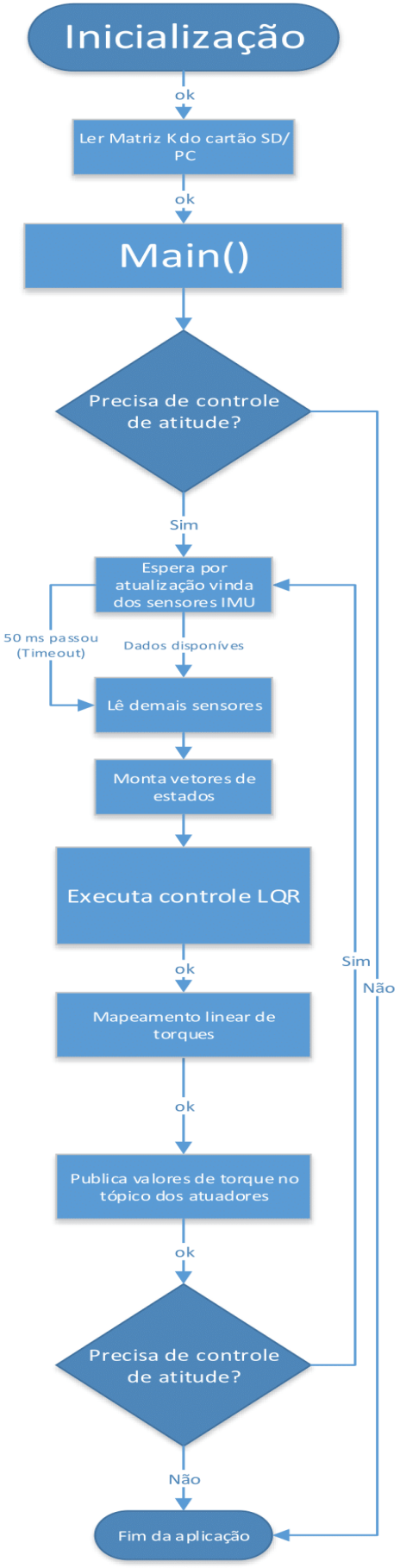

Figura 34 – Arquitetura da aplicação de controle LQR

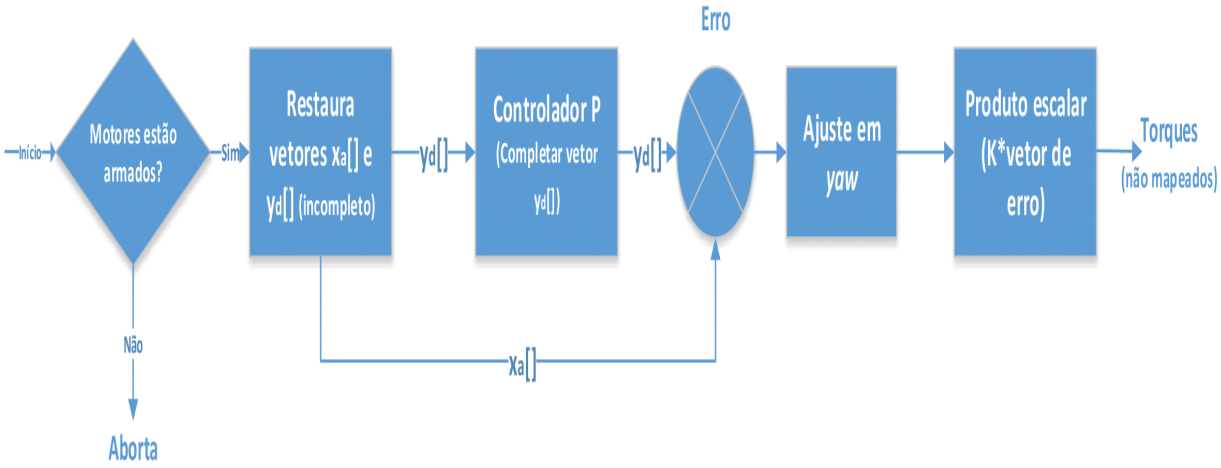

Figura 35 – Bloco de controle *LQR* implementado em *software*

5.3.2 Bloco *LQR*

Durante o desenvolvimento deste bloco, decidiu-se adotar as mesmas restrições temporais do estudo de caso 1, conforme Figura 32. Desejou-se, desde o início, em executar o *loop* de controle em uma fatia de tempo do período de amostragem do módulo *EKF*.

O bloco de controle *LQR* (Figura 35) checa primeiramente se os motores estão armados, caso o contrário, não há necessidade de execução de controle. Em seguida, restaura-se o vetor de estados desejados $y_d[\phi_d, p_d, \theta_d, q_d, \psi_d, r_d]$ e o vetor de estados atuais $x_a[\phi, p, \theta, q, \psi, r]$. Um detalhe é que não existem valores desejados para as velocidades (*p*,*q*,*r*), o módulo *Position Control* calcula apenas *setpoints* de ângulos.

Conforme é explicado na análise dos resultados, o estudo de caso 1 revelou uma grande necessidade de controle das velocidades angulares. De maneira à obter um valor desejado para as velocidades nos 3 graus de liberdade, foi implementado um controlador proporcional. Uma vez que os vetores estão completos, subtrai-se o estado desejado pelo estado atual e obtêm-se o vetor de erro.

Antes de calcular os torques, deve-se ajustar *yaw* para percorrer o menor caminho desejado, visto que este foi mais um problema identificado na implementação do controlador *PID*. Para ilustrar este problema, imagine que o veículo esteja em 0° *yaw* no *frame* inercial e recebe uma posição de 270° em sentido anti-horário para ser alcançada. Com esta função, o controlador consegue identificar que o caminho mais curto é efetuar 90° em sentido horário.

Por fim, é efetuado o cálculo dos torques multiplicando-se a matriz de ganhos cal-

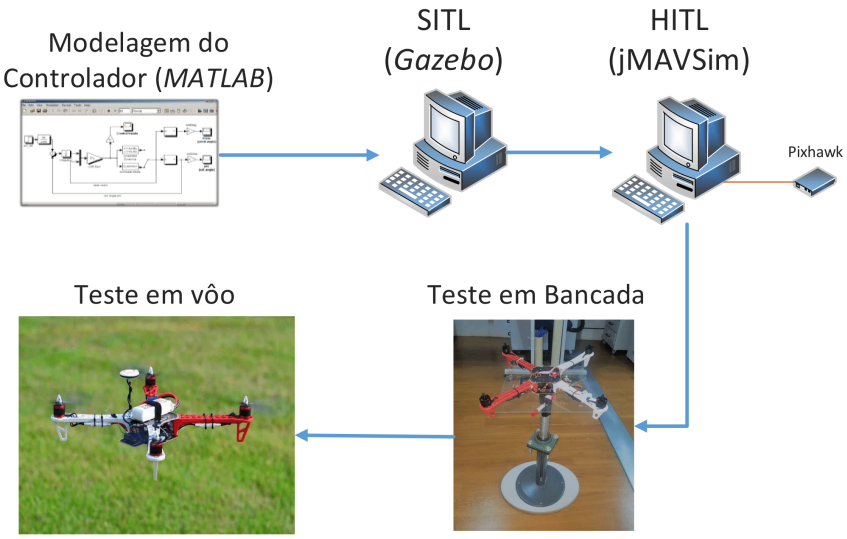

Figura 36 – Metodologia para implementação das estratégias de controle.

culada pelo vetor de erro. Os valores de torque serão mapeados antes de serem publicados para a aplicação do *Mixer*.

5.4 Resultados Experimentais e Análise

5.4.1 Procedimentos

Para a implementação dos estudos de caso, foi desenvolvida uma metodologia para que se pudesse avaliar o controlador de uma maneira segura. Esta metodologia está representada na Figura 36. A primeira etapa de modelagem do controlador em *software* *MATLAB* não será detalhada pois não foi escopo deste projeto, porém, esta parte foi desenvolvida por outros alunos e professores, e possuiu importância fundamental no decorrer deste trabalho.

A segunda etapa foi a implementação do controlador em *SIL*. Essa foi a segunda abordagem devido à praticidade de poder executar o controlador utilizando apenas o ambiente simulado. Foi escolhido o simulador *Gazebo* para testes em *SIL*, por este ser mais complexo e ser constantemente atualizado, além de possuir diversas funcionalidades interessantes como a aplicação de perturbações nos eixos x,y,z.

Uma vez que resultados considerados satisfatórios foram obtidos, testou-se o comportamento do controlador em *HIL*. Para esses testes usou-se o simulador *jMAVSim* por este ser a única opção disponível para veículos do tipo quadrirotor.Uma vez que os testes em *HIL* foram executados com sucesso partiu-se para os testes práticos.

Para os testes práticos, uma plataforma montada no laboratório conforme Figura 37, foi o ponto de partida. Esta plataforma fixa os braços do quadrirotor de maneira

Figura 37 – Plataforma de teste

em que este não consegue ganhar altura, mas é possível ver a resposta dos comandos de torque nos ângulos de Euler. Uma vez verificado que os comandos estão respondendo bem em plataforma de teste, os testes em vôo são realizados.

Desde o começo deste trabalho, sabia-se que esta abordagem para os procedimentos dos testes não é a ideal pelo fato de que cada etapa possui um modelo diferente. O modelo do quadrirotor do simulador *Gazebo*, o modelo do simulador *JMAVSim* e o modelo do quadrirotor *F450* usado na prática são diferentes. Porém, como o objetivo do trabalho é implementar dois controladores robustos (*PID* e *LQR*), e que não são fortemente dependentes de um modelo, tomou-se esta abordagem à princípio.

Vale ressaltar que tomou-se como base o modelo do *frame F450*, parcialmente identificado, para todas as etapas de desenvolvimento deste trabalho. O *kit* deste quadrirotor usado está disponível em [2].

5.4.2 Controlador PID

Uma vez que o código do controlador foi implementado, os testes em *SIL* utilizando o quadrirotor *Iris* no simulador *Gazebo* foram iniciados. Inicialmente, ajustar os ganhos foi uma tarefa árdua pois o quadrirotor do simulador simplesmente caía, sendo difícil analisar em qual dinâmica os ganhos não estavam respondendo bem.

Ainda, não se tinha certeza sobre os valores de torques que o controlador produzia. Conforme ressaltado, os torques do controlador passam pelo *mixer*, que força os comandos

Figura 38 – Quadrirotor *F450* utilizado neste trabalho (sem bateria).

SITL	ROLL	PITCH	YAW
P	1	1	0.01
I	0.0012	0.0012	0.001
D	0.04	0.04	0.0001

HITL	ROLL	PITCH	YAW
P	2.5	2.5	1.2
I	0.01	0.01	0
D	0.8	0.8	1

Figura 39 – Ganhos utilizados para o controlador *PID* em *SIL* e *HIL*

de torque à estarem entre -1 e 1.

Portanto, para melhor identificar o que estava acontecendo, adicionou-se uma funcionalidade de *debug* no código para imprimir na tela as importantes informações sendo calculadas pelo controlador. Esta abordagem só funciona para o simulador pois o *hardware* executa um sistema operacional de tempo-real que não consegue garantir esta funcionalidade à cada *loop* de controle.

Desta maneira, depois de uma repetitiva análise conseguiu-se encontrar ganhos que estabilizavam o veículo. A resposta do controlador em *SIL* para comandos de ângulo dados através do software de navegação pode ser vista na Figura 40.

A próxima etapa dos testes foram realizados em *HIL*. Estes testes foram importantes para avaliar o desempenho do controlador sendo executado diretamente no *hardware*, porém, o simulador disponíveis para teste, o *jMAVSim*, não proporciona uma segurança em relação ao ajustes dos ganhos do quadrirotor. Portanto, nesta fase não se perdeu

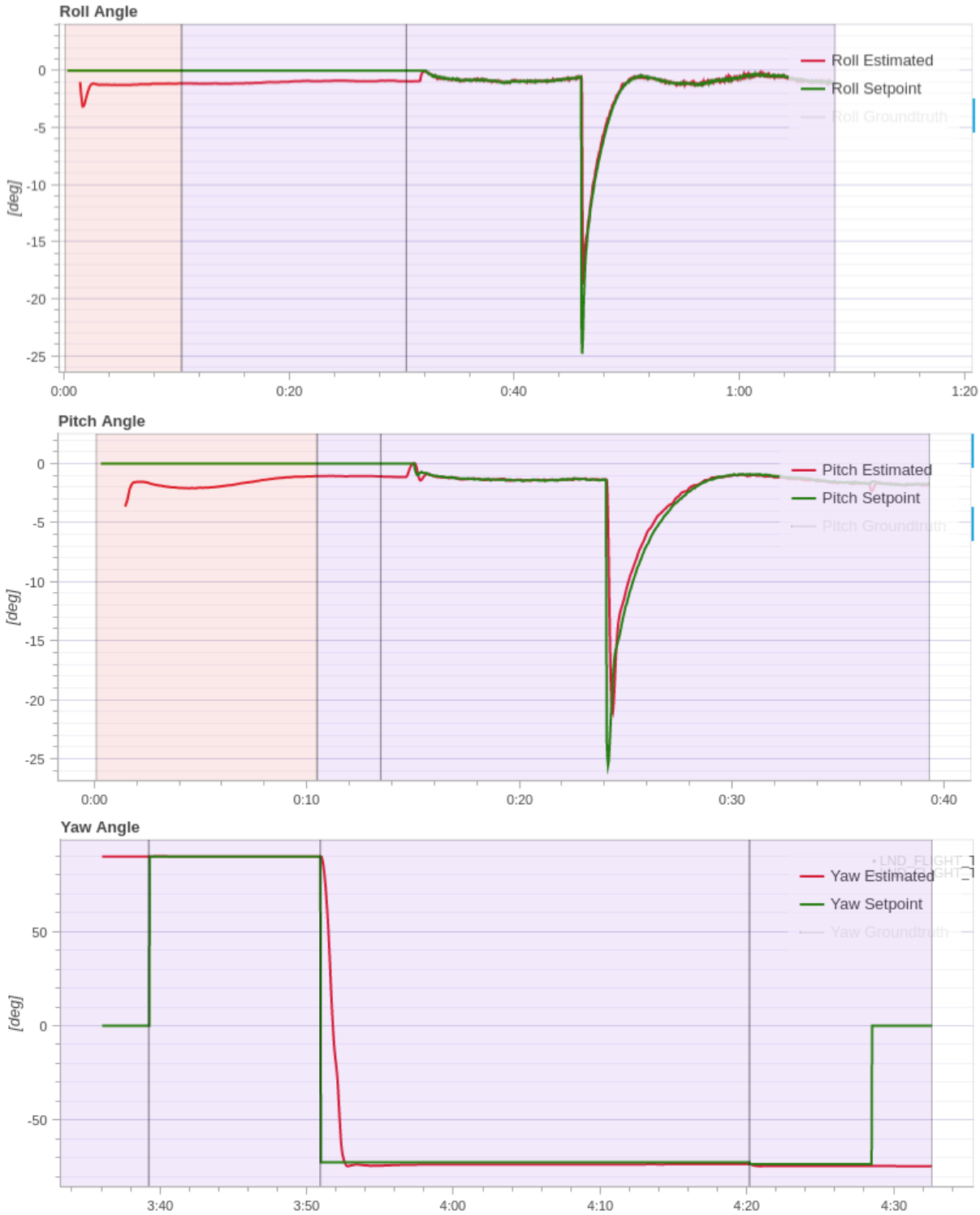

Figura 40 – Respostas do controlador *PID* em *SIL* para *roll*, *pitch* e *yaw*. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

muito tempo com o ajuste dos ganhos, e o foco foi em analisar a resposta do *hardware* aos comandos dados. É possível ver de acordo com a Figura 41 que a resposta do controlador acompanha os *setpoints* dados (cor verde), e este estabiliza bem no simulador não apresentando nenhuma vibração visível. Porém, acredita-se que esta resposta poderia ser ainda melhor depois de um ajuste fino nos ganhos.

O controlador *PID* apresentou resultados satisfatórios na bancada de teste. Com o auxílio do controle de rádio, foram dados comandos com o veículo armado preso à plataforma, e pode-se ver que o controlador estava respondendo bem. Uma vez que foi dado o devido ajuste nos ganhos em plataforma, partiu-se para a etapa de teste em vôo.

Entretanto, os resultados para teste em vôo não foram satisfatórios. Os ganhos foram ajustados, mas o veículo, ao decolar, não conseguia estabilizar e diversas quedas ocorreram. Depois de alguns componentes danificados, resolveu-se investigar algumas arquiteturas já existentes de *PID* para quadrirotor, como a da *PX4*, *MatrixPlot*, *Ardupilot*, entre outras.

Verificou-se que, em todos os casos investigados, os controladores *PID* existentes nestas plataformas utilizavam-se de controladores *PID* em cascata, conforme o controlador de atitude da *PX4*, cuja arquitetura foi discutida neste trabalho. Ao incorporar dois controladores *PID*, um para o controle angular e outro para o controle de velocidade angular, cada controlador trabalha com apenas um nível de integração[7], e assim o ajuste se torna mais simples. Existem sensores que medem a velocidade angular do veículo, fechando esta malha de controle, então, ao utilizar apenas um *PID* perde-se esta realimentação, e portanto, o ajuste é extremamente díficil. Entretanto, isto não significa que esta arquitetura de controle não funcione na prática, apenas ressalta que é necessário severos testes práticos para ajuste de ganhos, o que não era o objetivo principal deste trabalho.

O controlador *PID* implementado, mostrou-se que, mesmo realizando chamadas à funções que utilizam *drivers* para realizarem operações de *I/O*, funciona de maneira determinística, respeitando as restrições de tempo, e que possui um tempo de execução de *loop* menor do que o controlador padrão da *PX4* para todas as etapas do teste, o que será discutido adiante na comparação entre as aplicações de controle. Sua implementação em termos de recursos computacionais e cumprimento de *deadlines* foi concluída com sucesso em todas as etapas de teste.

5.4.3 Controlador *LQR*

Os testes iniciais do controlador *LQR* em *SIL* mostraram resultados muito ruins. Através de uma função de *debug*, conforme discutida nos resultados do controlador *PID*,

[7] Ângulo para velocidade angular, e velocidade angular para torque, ou seja, um nível de integração para cada estágio de controle.

Figura 41 – Respostas do controlador *PID* em *HIL* para *roll*, *pitch* e *yaw*. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

constatou-se que os torques gerados pelo controlador *LQR* estavam divergindo muito dos torques que o *mixer* trabalha (entre -1 e 1), ou seja, os motores saturavam rápido.

A função de mapeamento de torques resolveu bem este problema e o simulador apresentou resultados interessantes (Figura 42), considerando que o modelo do simulador é diferente do modelo identificado para cálculo dos ganhos usados para controle.

Os resultados dos testes em *HIL* foram também satisfatório como podem ser vistos na Figura 43, não havendo necessidade de alterar a matriz de ganhos. Obviamente, um ajuste nestes ganhos resultariam em uma melhor resposta, porém, como já ressaltado, o simulador *jMAVSim* não proporciona uma resposta muito confiável. Ao observar que o controlador respondia bem executando no *hardware*, deram-se início aos testes práticos.

Os testes em plataforma do controlador *LQR* mostraram que este tinha bastante potencial para um teste em vôo. Uma boa resposta aos comandos dados foi observada, e portanto o teste em vôo foi realizado.

Inicialmente, os testes em vôo não apresentaram bons resultados, observou-se que o veículo buscava executar ações muito bruscas. Decidiu-se portanto, por suavizar os torques através do mapeamento, fazendo testes experimentais até chegar ao valor de 30 $N \cdot m$[8], que pareceu ser uma boa referência para o controlador. Estes testes foram realizados com os motores do veículo armado, dando-se comandos de *roll* e *pitch* e observando se a resposta era agressiva ou fraca, procurando-se um ponto de equilíbrio que permitisse vôo. A matriz abaixo mostra a matriz de ganho K que foi utilizada para o teste em vôo, calculada através do algoritmo desenvolvido em *MATLAB*:

$$K = \begin{bmatrix} 1.2746 & 2.3401 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.5693 & 2.9010 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5941 & 1.8834 \end{bmatrix} \tag{5.2}$$

Na Figura 44 pode-se ver a resposta das dinâmicas em um dos vôos realizados com o controlador *LQR*. Vale ressaltar que as oscilações observadas nos gráficos são bem maiores em relação às observadas nos simuladores devido à diversos fatores. O primeiro deles é o fato de que o teste é executado no modo de vôo *POSITION*[9], isto implica que as referências passadas pelo rádio sejam de velocidades desejadas em (x,y), assim não é possível controlar os *setpoints* de ângulos gerados. O segundo fator é o ambiente de teste, que é um local bem aberto e com bastante vento, ambiente que dificulta a ação do controlador de se manter no ponto de operação. O terceiro é o fato de que sensores reais exibem mais erros do quê sensores simulados. Por último, o fator mais crucial é

[8] Este valor significa que o controlador sempre calcula torques entre -30 e 30 $N \cdot m$, e é mapeado entre -1 e 1 para o *mixer*.

[9] Este modo de vôo da *PX4* funciona da seguinte maneira: a altura é fixada, e o rádio envia comandso em (x,y).

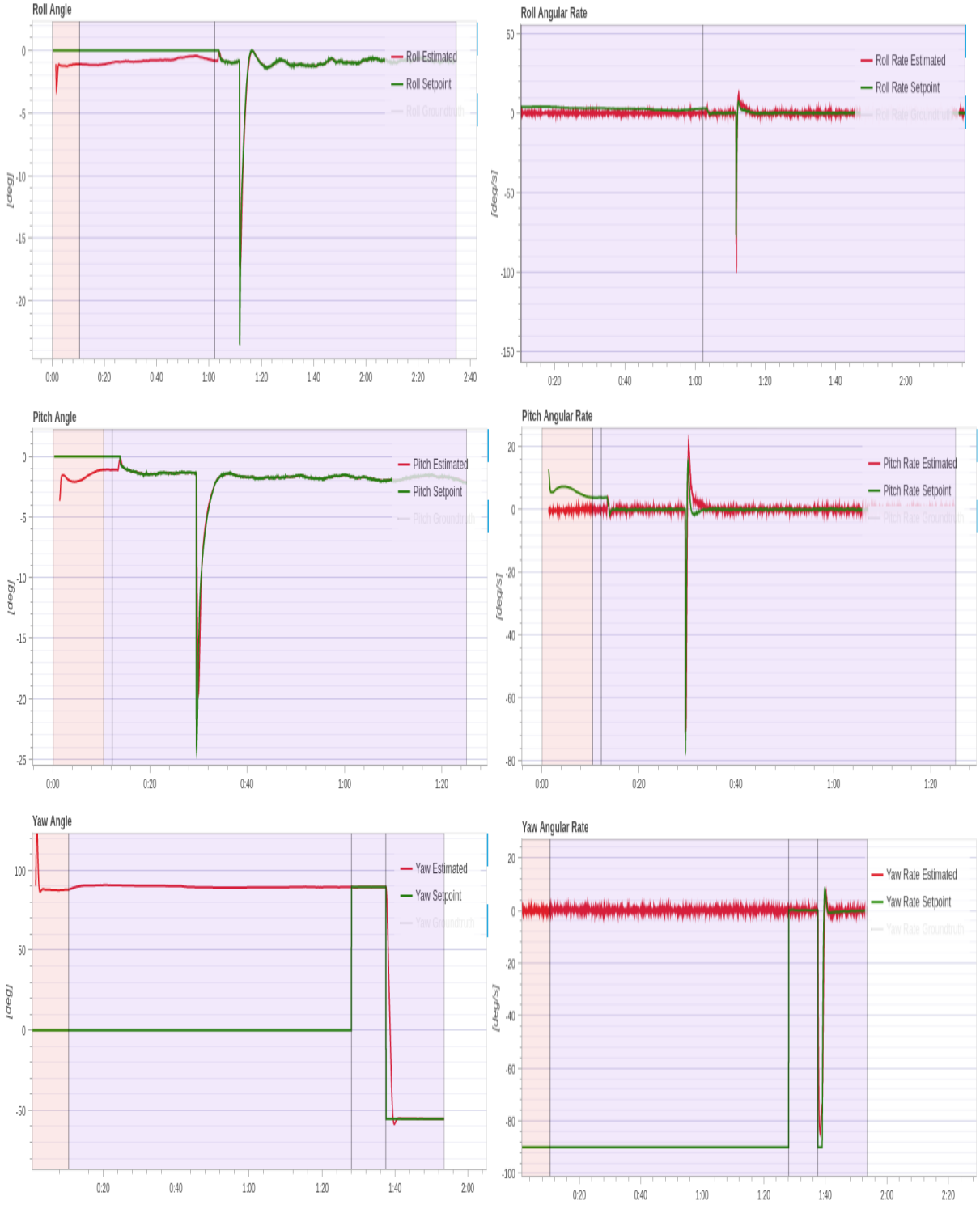

Figura 42 – Respostas do controlador *LQR* em *SIL* para *roll*, velocidade de *roll*, *pitch*, velocidade de *pitch*, *yaw* e velocidade de *yaw*. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

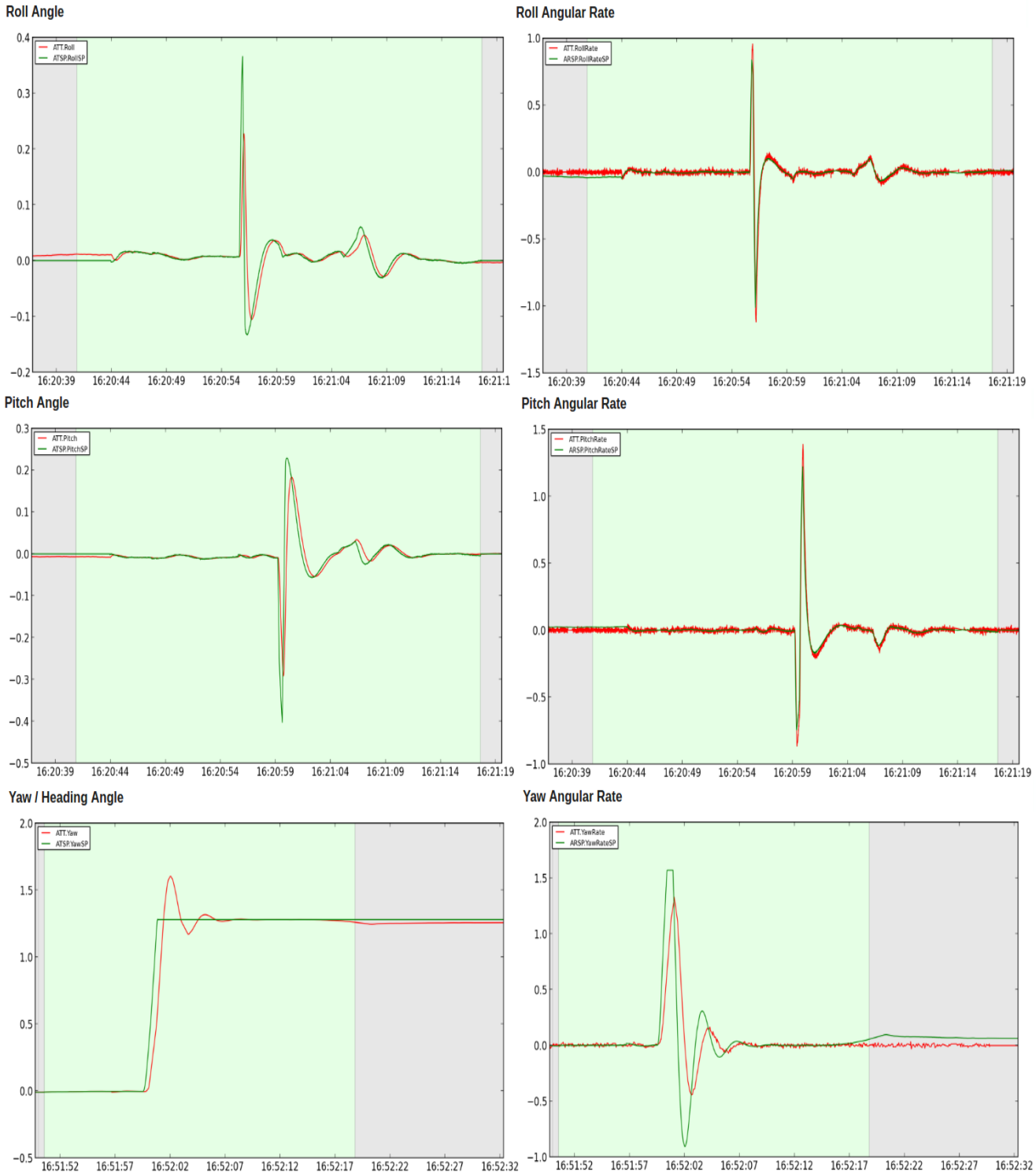

Figura 43 – Respostas do controlador *LQR* em *HIL* para *roll*, velocidade de *roll*, *pitch*, velocidade de *pitch*, *yaw* e velocidade de *yaw*. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

Software-in-the-loop

Aplicação	Tempo de teste	Tempo mínimo de loop	Tempo máximo de loop	**Tempo médio de loop**
attitude control (PX4)	9min e 32s	2μs	9156μs	17μs
controlador *PID*	9min e 58s	2us	7696μs	8μs
controlador *LQR*	10min e 15s	2μs	6145μs	9μs

Tabela 2 – Tempo de execução das aplicações em *SIL*

que não foram realizados severos ajustes nas matrizes de ganho, basicamente usou-se os valores que funcionaram bem no simulador, um ajuste fino nos ganhos proporcionaria um resultado melhor.

Na Figura 45 pode-se ver uma imagem fotografada durante um dos testes em vôo. O vídeo disponível em [7] também mostra o resultado alcançado em um dos vôos. O quadrirotor obedece bem os comandos. Na Figura 46 é possivel observar que o controlador consegue seguir bem a trajetória em x e y, e este gráfico ilustra, portanto, o ótimo desempenho do controlador *LQR*, uma vez que é necessário um bom controlador de atitude para o controle de posição.

5.4.4 Comparação dos controladores implementados com o controlador da *PX4*

Os controladores implementados demonstraram capacidade de executar respeitando as restrições de tempo. Vale ressaltar que para ambos o período de amostragem à cada *loop* de controle é de 4 ms, pois o estimador de estados da *PX4* amostra nesta frequência (250 Hz). Era de interesse, desde o começo do desenvolvimento, executar o controlador em uma fatia de tempo entre as amostras do estimador (apesar de que o quadrirotor provavelmente responderia bem em frequências menores), pois assim poderia considerar um controlador livre de latência.

A Tabela 2 apresenta informações de teste realizado no simulador em *SIL*. O computador utilizado para teste possui as seguintes configurações: processador *x64 Intel Core i7 2.0* GHz, 8 GB de mémoria *RAM* e sistema operacional *Ubuntu 16.04*. Como era de se esperar, os *loops* de controle executam em um tempo bem pequeno devido à alta disponibilidade de recursos. Os controladores *PID* e *LQR* ambos executam em uma parcela de tempo de médio muito próxima, ambos próximos da metade do tempo médio do controlador padrão da *PX4*. Entretanto, é interessante notar que o tempo máximo de *loop* mensurado ultrapassou os 4 *ms* desejados para todos os controladores, gerando latência neste sistema. Isto ocorre, pois o sistema operacional *Ubuntu* não é um sistema operacional de tempo-real, e portanto, não é determinístico.

A Tabela 3 mostra os tempos de execução das aplicações executando no *hardware Pixhawk*, em *HIL*. Pode-se notar que os controlador *PID* executa em um tempo médio

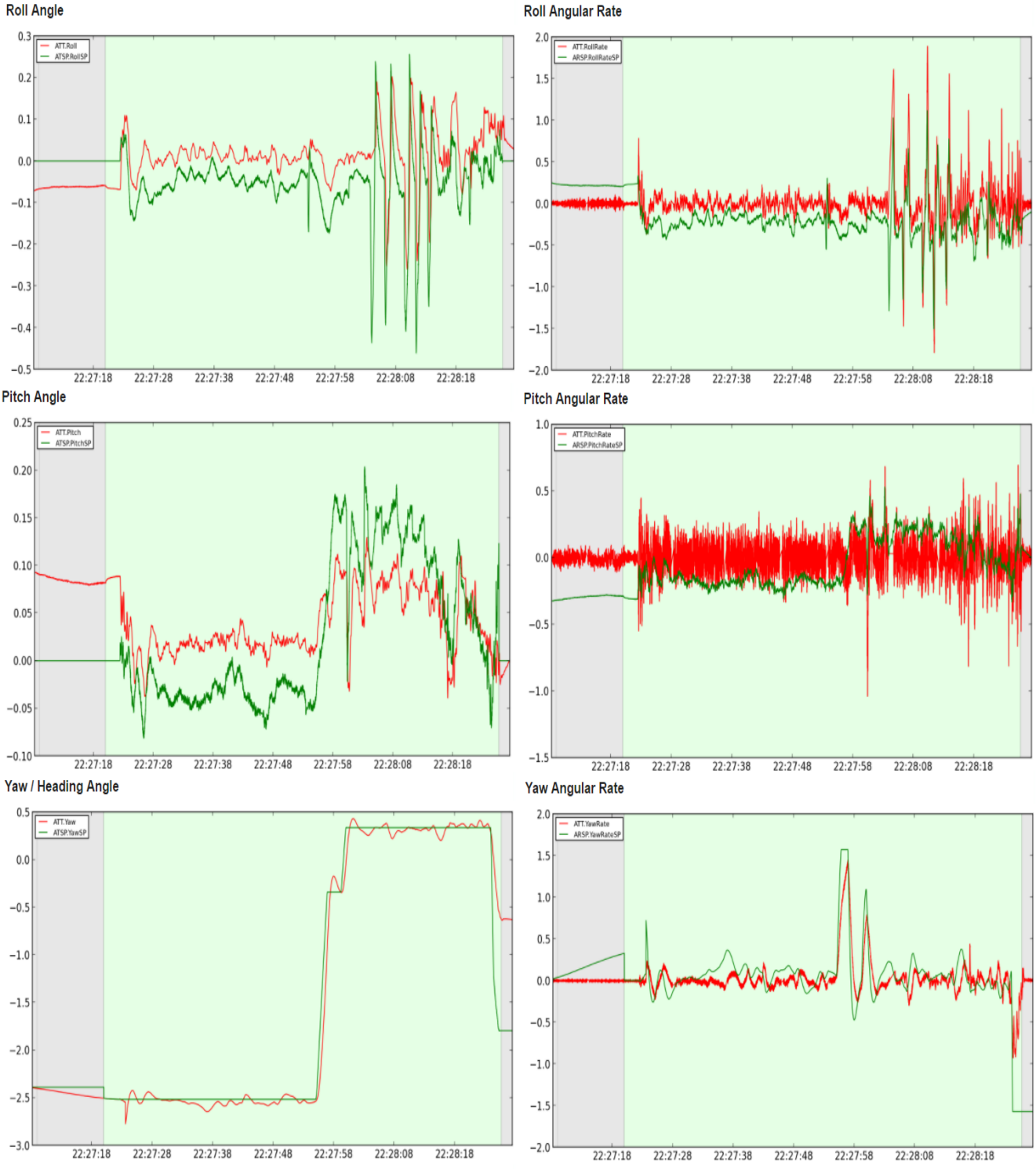

Figura 44 – Respostas do controlador *LQR* de um teste em vôo para *roll*, velocidade de *roll*, *pitch*, velocidade de *pitch*, *yaw* e velocidade de *yaw*. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

Figura 45 – Quadrirotor em vôo com a aplicação de controle LQR embarcada.

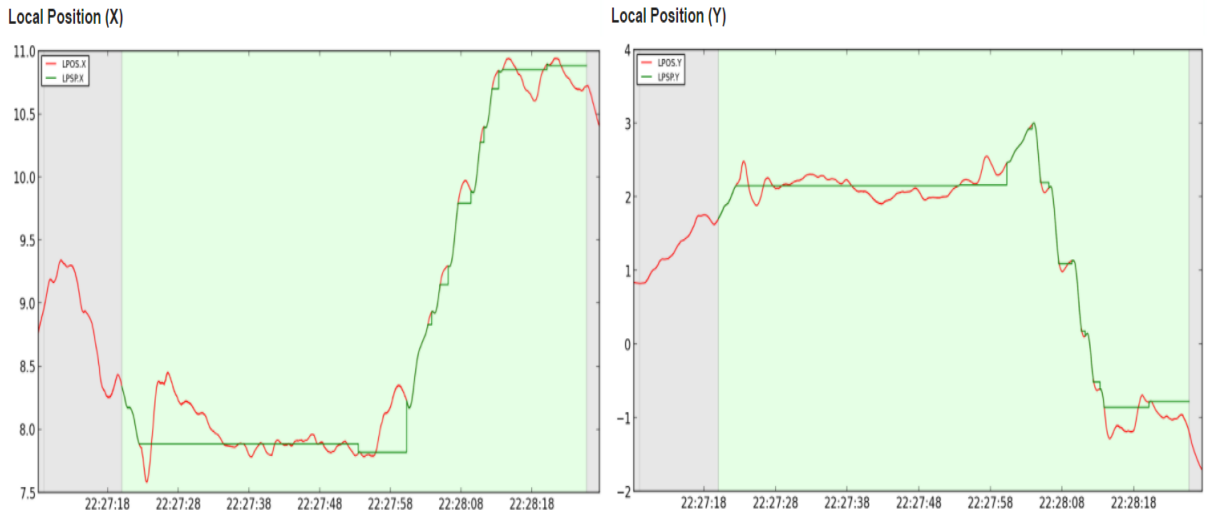

Figura 46 – Respostas do controlador LQR de um teste em vôo para as trajetórias em x e y. Em verde, o *setpoint* dado e em vermelho a resposta do sistema.

Hardware Pixhawk

Aplicação	Tempo de teste	Tempo mínimo de loop	Tempo máximo de loop	**Tempo médio de loop**
attitude control (PX4)	8min e 46s	25μs	498μs	136μs
controlador *PID*	9min e 15s	34us	393μs	43μs
controlador *LQR*	9min e 11 s	52μs	355μs	67μs

Tabela 3 – Tempo de execução das aplicações no *hardware*

menor que um terço do tempo médio de execução do controlador oficial da *PX4*. O *LQR* por ser um pouco mais complexo devido à multiplicação de matrizes, é um pouco mais lento que o controlador *PID* implementado, mas ainda assim, executa em menos de 50% do tempo do controlador oficial da *PX4*. Vale notar que mesmo os *loops* com maior tempo medido não ultrapassaram a marca de 0.4 ms, o que é 10% da fatia de tempo entre a amostragem de dados do estimador. Isto implica que ambas as aplicações são livres de latência, e executam sem sobrecarregar o sistema.

6 Conclusão

Conforme observado na introdução deste trabalho, os quadrirotores são veículos robóticos desenvolvidos no começo do século XXI cujas aplicações estão em diversas áreas seja com objetivo militar, de consumo ou comercial. O avanço da eletrônica possibilitou que estes veículos pudessem ser desenvolvidos por um baixo custo, o que influenciou diversos engenheiros, hobistas e pesquisadores à desenvolver soluções utilizando estes veículos.

Para que estes veículos sejam capazes de voar, o sistema de processamento destes devem ser capazes de cumprir com restrições de tempo, sendo usado sistemas operacionais de tempo-real para este propósito. Implementar uma estratégia de controle em um veículo quadrirotor requer um estudo da plataforma embarcada utilizada, para a utilização dos recursos existentes de maneira à atingir performance máxima para atender os *deadlines* propostos.

Este trabalho trouxe à proposta de implementar dois tipos de controladores utilizando-se da *PX4 Autopilot* uma plataforma de *hardware* e *software* existente para *VANTs*. Implementar o próprio módulo de controle nesta plataforma possibilita o entendimento de uma das soluções mais usadas por pesquisadores na área de *VANTs*, além de poder contribuir com outros estudantes da área, que podem através deste trabalho ter um caminho mais fácil já que a falta de documentação é um dos maiores problemas da *PX4*.

Os resultados alcançados no fim deste trabalho foram satisfatórios considerando toda a base que teve de ser construída no decorrer do projeto. O estudo do modelo matemático do quadrirotor, sistemas de controle, arquiteturas computacionais de quadrirotores e modelos de sistemas embarcados foi necessário para entender este completo sistema robótico. O estudo da *PX4* foi também uma tarefa bastante àrdua, pois em diversos pontos não havia documentação, apenas códigos-fonte à serem lidos para entender módulos necessários para implementar a aplicação de controle.

Os controladores propostos foram implementados com sucesso na plataforma *PX4*, respeitando restrições temporais e com respostas satisfatórias aos comandos tanto em *software-in-the-loop*, *hardware-in-the-loop* e testes em vôo, apesar de que poderia ter sido dedicado mais tempo aos ajustes finos para uma melhor estabilização dos controladores. O controlador *PID* implementado inicialmente no projeto, apesar de esta plataforma já possuir um controlador parecido, serviu para o entendimento do sistema *PX4*, e abriu o caminho para que o controlador *LQR* fosse implementado em um tempo bem menor do que o primeiro.

Por fim, para que os controladores propostos funcionassem de maneira mais autônoma e menos dependente da *PX4*, deveria ser implementado o *mixer* próprio com acesso

direto aos motores para que o controle não sofra regulações conforme mencionadas neste projeto. No apêndice é apresentado um guia de como acessar diretamente os motores e implementar o próprio *mixer*, e pode ser usado para dar continuidade à este projeto. Portanto, este trabalho fornece a base para outro pesquisador implementar a sua própria formulação de controle na plataforma *PX4 Autopilot*.

Ainda, outros trabalhos futuros interessantes seriam o desenvolvimento do modelo do quadrirotor *F450* em *SIL* e *HIL*, o que possibilitaria um melhor casamento entre os resultados do controle em cada etapa de implementação, e também, seria interessante a implementação de outras estratégias de controle.

Referências

[1] *Arduino Quadcopter*. http://hpcas.blogspot.com.br/. Visitado pela última vez em 19/06/2017. Citado 2 vezes nas páginas 15 e 34.

[2] *DJI E310 Kit*. http://www.dji.com/e310. Visitado pela última vez em 08/06/2017. Citado na página 88.

[3] *E305 420 Lite ESC*. https://store.dji.com/product/420-lite-esc. Visitado pela última vez em 16/04/2017. Citado 2 vezes nas páginas 15 e 49.

[4] *Explicando a Teoria PID*. http://www.ni.com/white-paper/3782/pt/. Visitado pela última vez em 10/06/2017. Citado 3 vezes nas páginas 15, 41 e 42.

[5] *Extended Kalman Filter and Control Library*. https://dev.px4.io/tuning_the_ecl_ekf.html. Visitado pela última vez em 15/04/2017. Citado na página 65.

[6] *How to choose the best battery for your drone*. http://www.dronetrest.com/t/lipo-batteries-how-to-choose-the-best-battery-for-your-drone/1277. Visitado pela última vez em 01/06/2017. Citado na página 50.

[7] *LQR controller embedded in Pixhawk hardware*. https://youtu.be/g66_KfxAI_Q. Visitado pela última vez em 18/06/2017. Citado na página 96.

[8] *MAVLINK Common Message Set*. https://pixhawk.ethz.ch/mavlink/. Visitado pela última vez em 01/04/2017. Citado na página 63.

[9] *PID controller*. https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/fas57_nyp7/Site/pidcontroller.html. Visitado pela última vez em 01/06/2017. Citado 2 vezes nas páginas 15 e 42.

[10] *PX4 Developer Guide*. https://dev.px4.io. Visitado pela última vez em 24/06/2017. Citado 2 vezes nas páginas 15 e 62.

[11] *PX4 Firmware Releases*. https://github.com/PX4/Firmware/releases. Visitado pela última vez em 19/04/2017. Citado na página 78.

[12] *PX4 Hardware*. https://github.com/PX4/Hardware. Visitado pela última vez em 23/05/2017. Citado na página 70.

[13] *PX4 Mixer Concept*. https://dev.px4.io/concept-mixing.html. Visitado pela última vez em 10/04/2017. Citado na página 68.

[14] *QGroundControl*. http://http://qgroundcontrol.com/. Visitado pela última vez em 15/06/2017. Citado na página 120.

[15] *Quadcopters, Multicopters: The Basic of Propellers and Motors*, 2015. Citado 2 vezes nas páginas 49 e 50.

[16] A. Gibiansky. *Quadcopter Dynamics, Simulation, and Control*. http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics/, 2012. Visitado pela última vez em 01/06/2017. Citado na página 73.

[17] Ardupilot Developer Team. *Mission Planner*. http://ardupilot.org/planner/docs/mission-planner-overview.html, Acessado pela última vez em 23/04/2017. Citado na página 119.

[18] A. Babushkin. *Simple multirotor simulator with MAVLink protocol support*. https://github.com/DrTon/jMAVSim. Visitado pela última vez em 01/06/2017. Citado na página 70.

[19] BBC News. *Amazon makes first drone delivery*. http://www.bbc.com/news/technology-38320067, Dezembro 2016. Visitado pela última vez em 13/06/2017. Citado na página 29.

[20] S. Bouadbdallah, A. Noth, and R. Siegwart. *PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor*. Swiss Federal Institute of Technology, Lausanne, Switzerland, 2004. Citado na página 43.

[21] Tommaso Bresciani. *Modelling, Identification and Control of a Quadrotor Helicopter*. page 180, 2008. Citado 2 vezes nas páginas 33 e 35.

[22] S. E. da Costa. *Controlo e Simulação de um Quadrirotor convencional*. Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa, Portugal, 2008. Citado na página 43.

[23] R. Dolci. *Avaliação Metrológica de Girômetros MEMS e Sua Aplicação em Testes de Inclinação de Modelos de Embarcações*. Universidade Federal de Santa Catarina, Brasil, 2014. Citado na página 38.

[24] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems*. Universidade de Estugarda, Alemanaha, 2002. Citado na página 39.

[25] G. Nutt. *NuttX Real-Time Operating System*. http://nuttx.org. Visitado pela última vez em 01/04/2017. Citado na página 60.

[26] H. Siddique. *Defibrillator-carrying drones could save lives, research suggests*. https://www.theguardian.com/technology/2017/jun/13/

defibrillator-carrying-drones-could-save-lives-research-suggests, Junho 2017. Visitado pela última vez em 13/06/2017. Citado na página 29.

[27] HobbyElectronics. *Imagem de um transmissor RC.* http://www.hobbytronics.co.uk/image/data/projects/multicopter/hobbyking-radio.jpg. Visitado pela última vez em 01/04/2017. Citado 2 vezes nas páginas 15 e 48.

[28] National Instruments. Citado na página 54.

[29] J. Valvano. *Embedded Systems: Shape the world.* https://www.edx.org/course/embedded-systems-shape-world-utaustinx-ut-6-10x. Visitado pela última vez em 11/01/2017. Citado na página 112.

[30] S. Khatoon, D. Gupta, and L. Das. *PID and LQR Control for a Quadrotor: Modeling and Simulation.* 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2014. Citado na página 41.

[31] P. Kmiecik and G. Granosik. *Real-Time Operating Systems for Robotic Applications: A Comparative Survey.* Citado na página 30.

[32] T. Krajnik, D. Fiser, V. Vonasek, and J. Faigl. *AR-Drone as a platform for robotic research and education.* Communications in Computer and Information Science, 2011. Citado na página 33.

[33] H. Lim, J. Park, D. Lee, and H.J. Kim. *Build Your Own Quadrotor.* IEEE ROBOTICS AND AUTOMATION MAGAZINE, September 2012. Citado na página 118.

[34] R. V. Lopes, P. Santana, and J. Ishihara. *Model predictive control applied to tracking and attitude stabilization of a vtol quadrotor aircraft.* 2011. Citado na página 38.

[35] Lorenz Meier. *MAVLink.* http://qgroundcontrol.org/mavlink/start, Abril 2017. Visitado pela última vez em 01/05/2017. Citado na página 63.

[36] T. S. Lourenço, A. M. de A. Pinto, and R. V. Lopes. *Model Predictive Control Applied to a Quadrotor UAV.* 2016. Citado 2 vezes nas páginas 38 e 40.

[37] T. Luukkonen. *Modelling and control of quadcopter.* 2011. Citado 3 vezes nas páginas 15, 35 e 36.

[38] E. Malinen. *Fusion of Data from Quadcopter's Inertial Measurement Unit Using Complementary Filter.* LUT Institute of Energy Technology, Finlândia, 2015. Citado na página 40.

[39] L. Meier, D. Honneger, and M. Pollefeys. *PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms*. 2015. Citado 6 vezes nas páginas 15, 30, 58, 59, 60 e 63.

[40] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. 2013. Citado 4 vezes nas páginas 15, 52, 53 e 54.

[41] O Globo. *Drone na mira dos negócios: de seguro a rodovias*. https://oglobo.globo.com/economia/drone-na-mira-dos-negocios-de-seguro-rodovias-21404420, Maio 2017. Visitado pela última vez em 08/06/2017. Citado na página 29.

[42] O. Liang. *How to choose ESC for racing drones and quadcopters*. https://oscarliang.com/choose-esc-racing-drones/. Citado na página 49.

[43] O. Liang. *Quadcopter PID Explained*. https://oscarliang.com/quadcopter-pid-explained-tuning/. Visitado pela última vez em 15/07/2016. Citado na página 41.

[44] A. Oliveira. *Usados na pulverização, drones geram economia de 80produtor rural*. http://g1.globo.com/sp/ribeirao-preto-franca/agrishow/2017/noticia/usados-na-pulverizacao-drones-geram-economia-de-80-ao-produtor-rural.ghtml, Maio 2017. Visitado pela última vez em 13/06/2017. Citado na página 29.

[45] S. M. C. Patrao. *Ferramenta de Teste e Validação para Algoritmos de Fusão Sensorial*. Instituto Superior de Engenharia de Coimbra, Portugal, 2015. Citado 2 vezes nas páginas 39 e 40.

[46] PX4 Developer Team. *Flight Modes*. https://dev.px4.io/en/concept/flight_modes.html. Visitado pela última vez em 22/04/2017. Citado na página 79.

[47] PX4 Developer Team. *Multicopter PID Tuning Guide*. https://docs.px4.io/en/advanced_config/. Visitado pela última vez em 17/04/2017. Citado na página 77.

[48] PX4 Deveveloper Team. *About PX4 Autopilot*. http://px4.io/about-us/. Visitado pela última vez em 08/06/2017. Citado na página 30.

[49] S. Sharma. *Getting Started With AVR Microcontrollers*. https://techawarey.wordpress.com/2013/10/16/, 2013. Visitado pela última vez em 02/12/2016. Citado 2 vezes nas páginas 16 e 111.

[50] Inkyu Sa and P. I. Corke. *System identification, estimation and control for a cost effective open-source quadcopter*. Proceedings - IEEE International Conference on Robotics and Automation, 2012. Citado na página 33.

[51] P. R. Q. A. Santana and M. Braga. *Concepção de um veículo aéreo não-tripulado do tipo quadrirrotor*. Universidade de Brasília, Brasil, 2008. Citado 2 vezes nas páginas 35 e 81.

[52] PX4 Developer Team. *PX4 Firmware*. https://github.com/PX4/Firmware. Visitado pela última vez em 01/04/2017. Citado 3 vezes nas páginas 63, 64 e 68.

[53] The Economist. *Taking Flight: Civilian Drones*. http://www.economist.com/technology-quarterly/2017-06-08/civilian-drones, Junho 2017. Visitado pela última vez em 15/06/2017. Citado na página 29.

[54] J. W. Valvano. *Embedded Microcomputer Systems*. 2011. Citado 7 vezes nas páginas 16, 54, 112, 113, 114, 115 e 116.

[55] A. Zulu and S. John. *A Review of Control Algorithms for Autonomous Quadrotors*. Journal of Applied Sciences, 4:547–556, 2014. Citado na página 40.

Apêndices

APÊNDICE A – Primeiro Apêndice

A.1 Sistemas embarcados

Sistemas embarcados são sistemas em que existem um software embutido em um hardware dedicado para uma aplicação específica. O hardware é geralmente um microcontrolador, dispositivo que incorpora um processador, memória e portas de entrada e saída (E/S) em único pacote, possível de ser reprogramado. Os dispositivos de E/S são de extrema importância pois estes fornecem a necessária funcionalidade ao sistema, como motores e sensores. Um exemplo de sistema embarcado pode ser visto na Figura 47 .

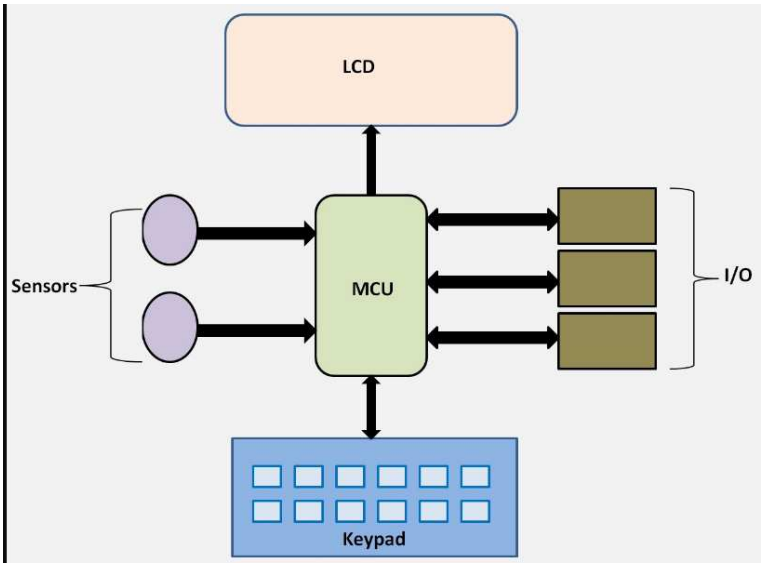

Figura 47 – Exemplo clássico de um sistema embarcado. Adaptado de [49]

Nesse exemplo podemos ver o microcontrolador representado pela sigla *MCU*. Esse dispositivo possui uma memória interna, recebe dados de sensores (*sensors*) e do usuário através do teclado (*keypad*), e têm como saída de dados o visor de LCD. O sistema pode possuir outras entradas e saídas como podemos ver nos blocosde E/S (*I/O*).

Os sistemas embarcados estão presentes em todo momento na sociedade atual. O carro é um exemplo de sistema que possui diversos sistemas embarcados embutidos. O sistema que controla o freio ABS e o sistema que monitora a temperatura do motor são exemplos. Sistemas embarcados podem integrar partes mecânicas, elétricas, químicas e computacionais em um só sistema.

Sistemas de tempo-real são sistemas que operam com restrições de tempo. Podem ser classificados de duas maneira: *soft real-time* ou *hard real-time*. O primeiro são sistemas em que existem tarefas com prioridades de execução e estas devem ser respeitadas, porém

uma falha não acarreta em comprometimento do sistema. Já o *hard real-time* é um sistema em que as restrições de tempo devem ser obrigatoriamente respeitadas, caso contrário uma falha grave ocorrerá. Essas restrições devem ser cumpridas mesmo em condições de sobrecarga do sistema, como múltiplas tarefas sendo executadas ao mesmo tempo [29] .

O quadrirotor é um sistema *hard real-time*, pois para que este funcione de maneira adequada deve existir uma garantia de entrega de dados no tempo. Os sensores devem ser lidos, seus dados processados, e a rotação dos motores deve ser regulada, tudo isso respeitando um *deadline* - intervalo de tempo entre a observação do estímulo e a reação do mesmo) - caso contrário uma falha catastŕofica poderá acontecer (o veículo cair, por exemplo).

A.1.1 Interfaces

Um dos fatores que diferencia sistemas embarcados de computadores pessoais são os dispositivos de E/S que são acoplados ao sistema embarcado [54]. Antes de falar um pouco sobre as interfaces com o dispositivos de entrada e saída, deve-se entender a importância de alguns parâmetros de performance quando se fala sobre interface com sistemas embarcados. Os parâmetros citados abaixos são fundamentais para avaliar o sistema de tempo-real em que se está trabalhando ou desenvolvendo:

- Latência: tempo entre o dispositivo de E/S precisar de um serviço e o serviço ser inicializado (inclui atrasos tanto de hardware quanto de software).
- Largura de banda (*throughput*): o fluxo de dados máximo que pode ser processado pelo sistema (bytes/segundo).
- Prioridade: determina a ordem em que será executadas tarefas quando duas ou mais solicitações são feitas simultâneamentes.

Um problema frequente relacionado aos dispositivos de E/S é que estes são bem mais lentos do que a execução de software pelo processador (ex: os dados de um sensor pode estar chegando a 1KHz, enquanto o software da *flight controller Pixhawk* é executado por um processador de 168 MHz). O software deve ser sincronizado com o hardware de maneira em que a comunicação seja efetiva.

A figura 48 ilustra uma situação comum. A CPU (rodando o software) espera até o dispositivo estiver pronto (sai do estado *busy* para *done*). Quando esse evento ocorre, a CPU executa uma rotina de leitura, e assim que termina envia um sinal para o hardware dizendo que já pode voltar a execução, enquanto que o software irá executar o processamento em cima dos dados lidos. Como a CPU executa muito rápido, após realizar o devido processamento a mesma deve esperar até o hardware estiver pronto novamente.

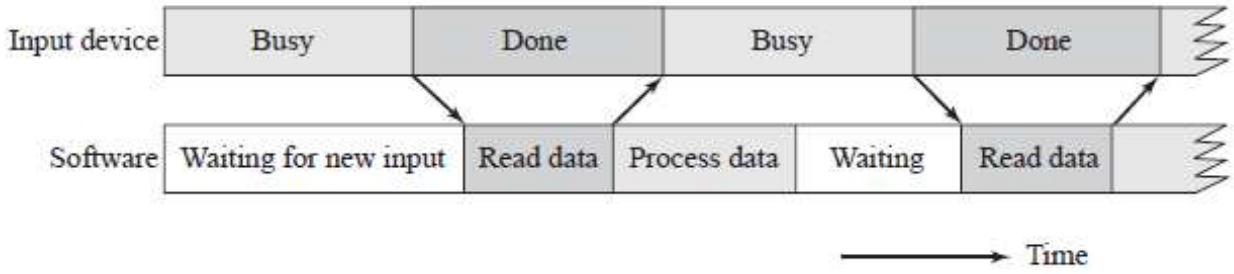

Figura 48 – Software esperando por hardware para realizar sua rotina. Adaptado de [54]

Existem alguns métodos para resolver o problema de sincronização, e os mais utilizados em sistemas embarcados são:

- *Blind cycle*: método em que o software espera um determinado tempo fixo e assume que o hardware está pronto e começa a executar rotinas de leitura/escrita. Esse método, que pode ser traduzido como "ciclo cego", é utilizado em situações em que é fácil predizer a velocidade do dispositivo conectado ao microcontrolador.

- Busy-wait: método em que o software espera pelo hardware até este estar pronto para realizar a comunicação. O software fica preso em um estado de espera até que receba um sinal do dispositivo de hardware dizendo que este pode realizar rotinas de E/S. Esse método é bom para sistemas mais simples, e claramente não é adequado para sistemas de tempo-real pois o sistema fica travado até receber uma resposta de um dispositivo de hardware.

- Interrupções: ferramenta utilizada pelo hardware para causar uma execução de uma rotina especial no software. Bastante utilizadas em sistemas de tempo-real, interrupções permitem o hardware enviar um sinal quando estiver pronto, e assim o software para instântaneamente e executa uma rotina de leitura/escrita. Assim, o software executa suas operações normalmente sem entrar em estado ocioso até receber uma chamada de interrupção.

- Periodic polling: Usa o sinal de clock da CPU para gerar interrupções periódicas e checar se o hardware está pronto para realizar rotinas de leitura/escrita. Método bastante utilizado quando o dispositivo de hardware não permite interrupções.

Para este trabalho, durante o desenvolvimento de aplicação para o sistema embarcado do quadrirotor, focou-se nos dois últimos métodos. Interrupções são importantes para um sistema como o do quadrirotor, pois o sistema estará executando múltiplas aplicações ao mesmo tempo, portanto, é necessário uma maneira de interromper a execução

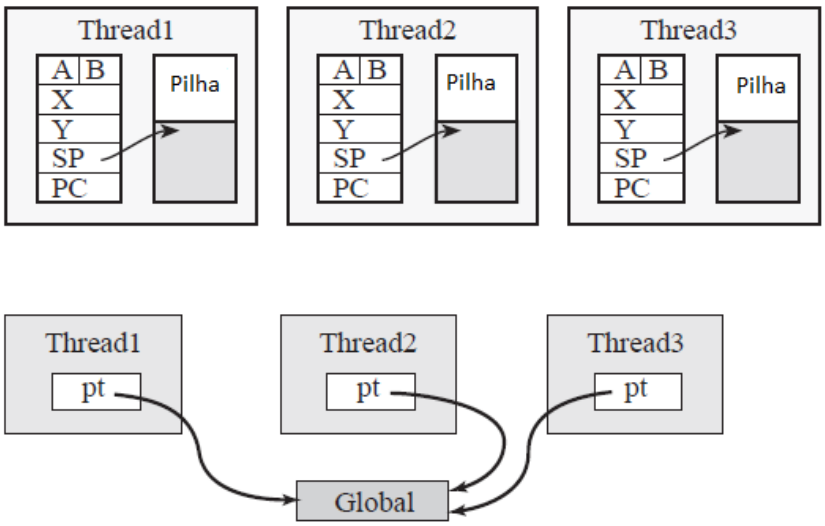

Figura 49 – *Threads* possuem registradores e pilhas próprias, porém podem compartilhar recursos. Adaptado de [54].

do software principal e sinalizar que alguma outra tarefa precisa ser executada, como por exemplo leitura dos dados de um sensor importante qualquer. O método *periodic polling* é bastante utilizado em sistemas de tempo-real como por exemplo, uma função de sistemas Unix que implementa, em partes, esse método, chamada *poll()*. Ela é bastante utilizada no software embarcado da *Pixhawk* para checar por novos dados de sensores periodicamente.

A.1.1.1 *Threads*

Uma *thread* é uma rotina de *software*. Pode ser usada por aplicações para dividir tarefas, ou seja, fazer com que múltiplas rotinas executem de forma concorrente ou paralela, caso seja um processador de múltiplos núcleos. As *threads* têm pilha e registradores próprios, o que significa que elas possuem estado próprio. Quando o processador decide executar uma nova *thread*, existe uma troca de contexto, ou uma troca de estados, onde os valores dos registradores são substitúidos pelos valores dos registradores da nova *thread*, assim começa-se a ler da pilha desta. As *threads* porém, compartilham variáveis globais e dispositivos de E/S, como pode ser visto na Figura 49.

Isso é bem interessante, pois significa que as threads podem executar suas rotinas e comunicar-se com outras através das variáveis globais. Pode existir uma *thread* que executa uma rotina de leitura de um sensor toda vez que esse hardware estiver disponível, e grava os valores lidos em uma estrutura de dados global, permitindo que uma outra *thread* de processamento tenha acesso aos dados.

Uma *thread* pode está em quatro estados representados na Figura 50. Em configurações bem simples, as *threads* só podem estar no estado executando ou ativa. A *thread* que está no estado executando, significa que ela está literalmente executando no presente

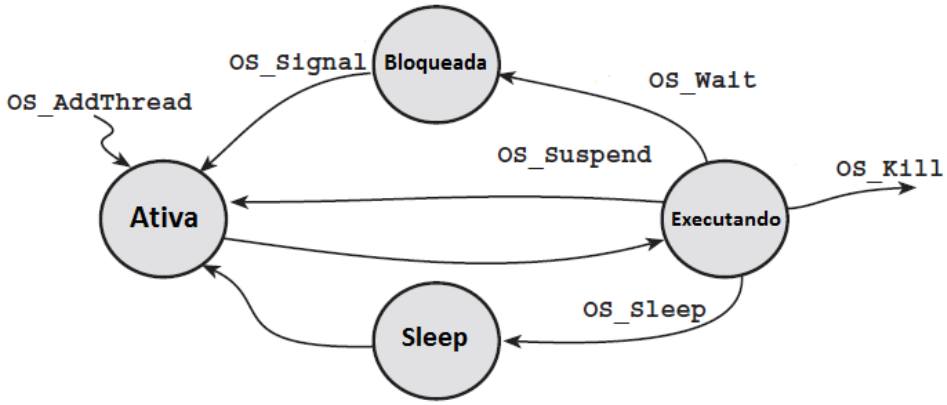

Figura 50 – Estados de uma *thread*. Adaptado de [54].

momento. Já no estado ativa, quer dizer que ela está pronto para ser executada, porém está esperando para rodar, já que em sistemas embarcados só existem processadores de um núcleo até o presente o momento, ou seja, só uma aplicação pode rodar por vez.

Uma *thread* no estado bloqueado signica que esta está esperando por algum evento externo para retomar sua execução. Um exemplo disso é quando uma *thread* deseja escrever caracteres em um display que já está sendo usado por outra *thread* no momento, desse modo a primeira fica em estado bloqueado até que o recurso esteja liberado. Os semáforos implementam a comunicação e sincronização entre as *threads*.

Por fim, a *thread* pode estar em estado *sleep* (dormindo). Isso acontece em aplicações em que deve-se esperar determinado tempo para que esta retome a execução. O sistema operacional coloca a *thread* para dormir, e acorda a mesma passado o tempo.

A Figura 50 mostra as transições possíveis entre os estados. A *thread* transiciona entre os estados, na maioria das vezes, fazendo chamadas a funções do sistema operaciona, representadas pelas palavras começando com *OS* na figura.

A.1.1.2 Escalonador

O escalonador é uma função do sistema operacional que executa as *threads* uma por uma. As *threads* que estão prontas para ser executadas (estado ativo) são listadas e quando o processador estiver livre, o escalonador escolherá uma delas para ser executada. Em um escalonador preemptivo, as *threads* são suspensas temporariamente por uma interrupção e o escalonador escolhe uma para ser executada. Já em um escalonador não-preemptivo, as *threads* decidem por si só quando irão parar e deixar outra executar. Escalonadores preemptivos são os mais adequados quando se trata de RTOS.

Uma das funções principais do escalonador é escolher qual *thread* será executada. Quando o escalonador decide interromper a execução de uma *thread* pela CPU, ele precisa salvar o estado atual desta *thread*, e antes de começar a execução da próxima, ele precisa

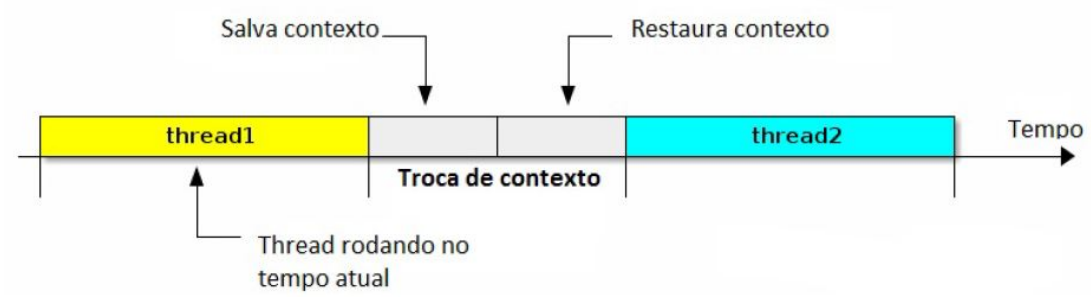

Figura 51 – Troca de contexto

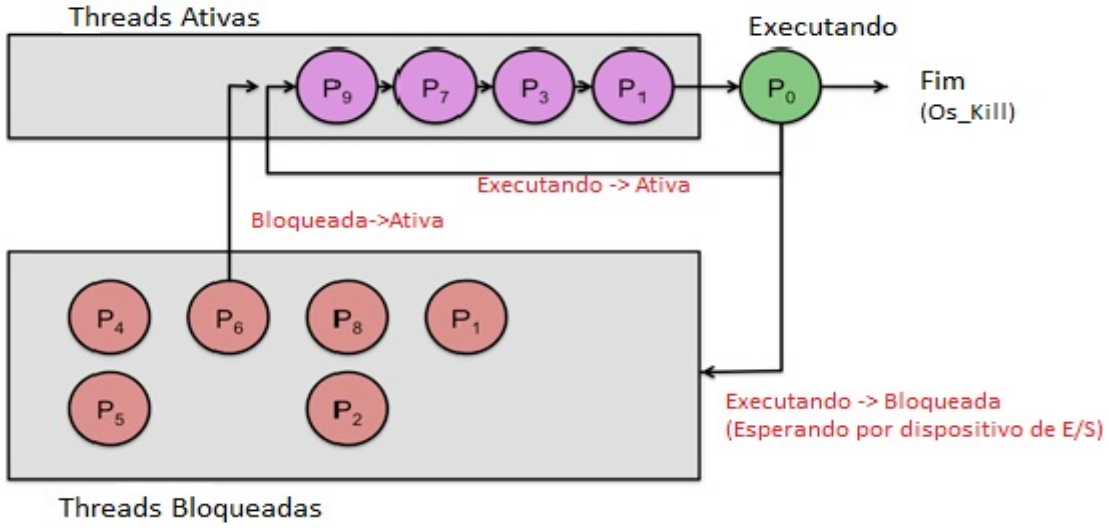

Figura 52 – Escalonador *Round-Robin* (RR). Escrito em vermelho podemos ver as possíveis transições de estados das *threads*. O escalonador executa as *threads* que se encontram em estado ativo de forma circular.

restaurar o estado em que esta *thread* estava antes de ser interrompida. Esse conceito é chamado de troca de contexto (Figura 51), e o tempo em que o sistema operacional gasta fazendo isso deve ser o menor possível em sistemas de tempo-real.

Escalonadores para RTOS têm métricas delimitadas para importância e prioridade de execução. Dependendo do tipo do sistema, o escalonador pode dar prioridade para *threads* que executam rotinas de leitura de um *hardware*. Pode ser que o sistema prefira dar prioridade para as *threads* que executam mais frequentemente. Isso tudo irá depender do tipo de sistema, e a melhor forma de cumprir com as *deadlines* destes. Dito isso, podemos generalizar os tipos de escalonadores mais comuns de acordo com [54]. São eles:

- ***Round-Robin***: executa as *threads* em estado ativa de uma maneira circular, dando à cada uma delas um tempo igual de execução Figura 52. Existem variações onde os tempos são dados de acordo com "pesos"atribuídos a cada *thread*.
- **Escalonador de prioridade**: determina um número que representa a prioridade de

execução de cada *thread*. Executa-se então as *threads* em estado ativa em ordem de maior prioridade para menor. Caso as *threads* tenham prioridade iguais, o sistema passa a ser executado como o *round-robin*.

- **Fila exponencial**: escalonador dinâmico que varia as prioridades e os tempos de execução dados à cada *thread* à medida em que estas vão executando. O valor que será atribuído de prioridade depende do tipo de algoritmo utilizado para resolver esse escalonador.

- **Taxas Monotônicas (RMS)**: escalonador que atribui prioridades dependendo do quão frequente a *thread* executa. *Threads* que executam bastante ganham alta prioridade.

A.2 Placas controladoras de vôo comerciais

Existem diversas modelos de placas controladoras de vôo (*flight controllers*) no mercado, que são projetadas já com diversos componentes integrados como os sensores girômetros, acelerômetros, barômetros e *GPS*, além de circuitos de comunicação, antenas e transceptores. Essas placas controladoras são escolhidas de acordo com a aplicação do *VANT*, pois cada uma tem especificação de *hardware* e *software* diferente. Existem aplicações que necessitam, por exemplo, de muita memória para processamento, como aquelas que usam algoritmos de visão computacional, outras que necessitam de um alta memória de barramento para acessar múltiplos dispositivos simultâneamente, outras que possuem componentes de ajuste de ganhos de controle manuais.

Entre as controladoras de vôo existente no mercado, existem as de código-aberto, as quais qualquer usuário pode fazer o *download* dos arquivos fontes dos algoritmos de controle e modificá-los da maneira que achar conveniente, e existem também as que são de código fechado, as quais o sistema são propriedade dos seus criadores e os usuários não podem modificá-los.

Em um estudo de 2012 [33], o autor fez um levantamento das placas controladores de vôo cujo os projetos são abertos ao público, com base no volume de usuários e quão ativo está o projeto na comunidade. Com base nisso, ele fez um *ranking* das mais relevantes que são:

- *Arducopter* (Figura 53-a): baseado na plataforma *Arduino* e desenvolvido por engenheiros individuais ao redor do mundo, fornece software com interface gráfica que exibe informações de vôo e ajuste de ganhos para o controlador.

- *Openpilot* (Figura 53-b): utiliza licensa *General Public License* (*GPL*), e o seu sistema possui um *RTOS* modificado do *FreeRTOS*. *Software* com interface gráfica é fornecido para ajuste de ganhos e recebimento de dados de vôo.

- *Paparazzi* (Figura 53-c): é um sistema de piloto automático desenvolvido para diversos tipos de *VANT*, com licensa *GPL*. Fornece software com interface gráfica para planejar missões de vôo.

- *Pixhawk* (Figura 53-d): sistema de piloto automático que possui algoritmos desenvolvidos para pilotagem através de imagens, e está sob licensa *GPL*. Fornece software com interface gráfica.

- *Mikrokopter* (Figura 53-e): sistema de piloto automático desenvolvido para quadrirotores. Fornece *software* com interface gráfica com ajustes de ganho para o controlador e monitoração de bateria.

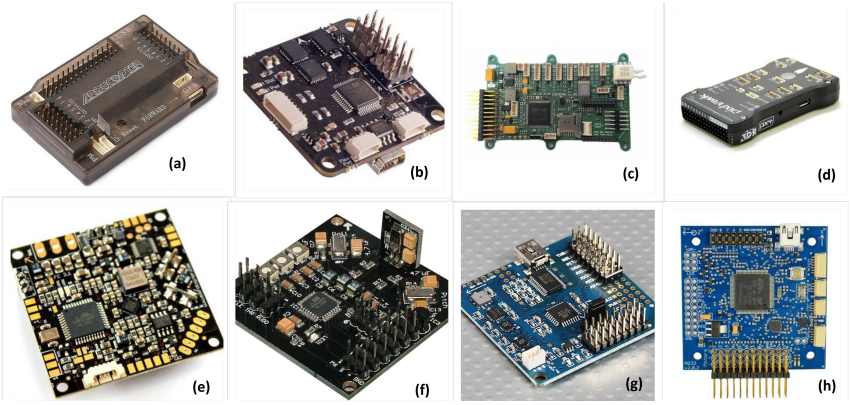

Figura 53 – (a)*Arducopter*; (b)*Openpilot*; (c)*Paparazzi*; (d)*Pixhawk*; (e)*Mikrokopter*; (f)*KKmulticopter*; (g)*Multiwii* e (h)*Aeroquad*

- *KKmulticopter* (Figura 53-f): sistema de piloto automático simples contendo um microcontrolador de 8 *bits* e um girômetro para cada eixo. Não fornece *software*.
- *Multiwii* (Figura 53-g): sistema de piloto automático para quadrirotores que utiliza o arduino como processador, e sensores que variam de acordo com o modelo. Está sob licensa *GPL* e é fornecido *software* com interface gráfica.
- *Aeroquad* (Figura 53-h): baseado em Arduino, é também um sistema de piloto automático desenvolvido para quadrirotores, e fornece *software* com interface gráfica.

Existem diversos outros modelos porém, que pequenos grupos desenvolvem baseados em plataformas de código-aberto como *Arduino* e adicionam dispositivos necessários de acordo com a necessidade da aplicação a ser desenvolvida.

A.3 *Softwares* de navegação

Os *softwares* de navegação são basicamente utilizados para guiar o veículo em modo de vôo automático e fornecer informações sobre o vôo e *status* do veículo.

Um dos *softwares* de navegação mais conhecidos é o *Mission Planner*. Abaixo, têm-se uma lista de funcionalidades deste *software* [17]:

- Gravar ultima versão de *firmware* disponível diretamente na *flight controller*.
- Configurar e ajustar os ganhos de controle do veículo
- Planejar, salvar e carregar missões autônomas para o hardware de maneira simples, apenas clicando nos pontos do mapa
- Geração de registros de missão, contendo informações dos vôos para análise

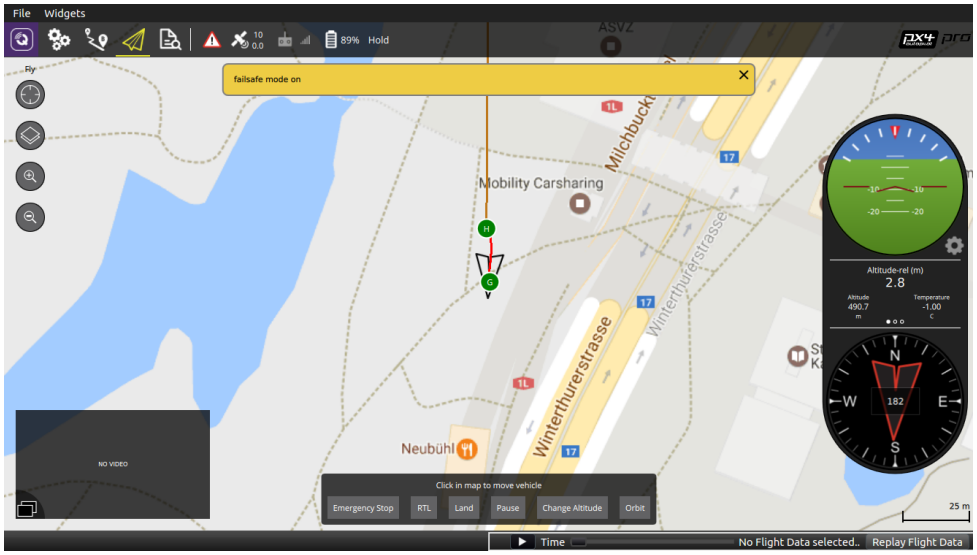

Figura 54 – *QGroundControl*: software de navegação que suporta o protocolo *MAVLink*

- Funções de telemetria para monitoração do estado veículo durante vôo

Outro *software* de navegação bastante famoso, é o *QGroundControl* [14]. Por ser compatível com os projetos *PX4 Autopilot* e *ArduPilot*, e suportar o protocolo MAVLink, é uma das plataformas mais utilizadas atualmente. Este *software*, por ser de código-aberto, também permite a implementação de funcionalidades personalizadas por parte dos desenvolvedores, o que faz o QGroundControl ser uma excelente escolha.

APÊNDICE B – Segundo Apêndice

B.1 Implementando um controlador na *PX4*

B.1.1 Configurando ambiente para *Pixhawk* e Simulador

Para colocar um algoritmo de controle próprio no *firmware* da *PX4* deve-se primeiro instalar as dependências e a *toolchain*. Para isto deve-se seguir o guia contido em https://dev.px4.io/en/setup/dev_env.html. Para este trabalho a parte de interesse está localizada em *NuttX based hardware*. Em seguida acessar https://dev.px4.io/en/setup/dev_env_linux_boutique.html#toolchain-installation e instalar a ultima versão da *toolchain* em *Toolchain Installation*.

À seguir deve-se clonar o repositório e compilá-lo. Para isso deve-se digitar no terminal:

```
$ mkdir -p ~/src
$ cd ~/src
$ git clone https://github.com/PX4/Firmware.git
$ cd Firmware
$ git submodule update --init --recursive
$ cd ..
```

Após isso o ambiente está configurado e já pode-se compilar o *firmware* da *PX4*. Digite na linha de comando:

```
$ cd ~/src/Firmware
$ make px4fmu-v2_default
```

Esse código irá compilar o *firmware* para a *Pixhawk*. Caso queira testar se o modo de simulação está funcionado deve-se digitar no terminal:

```
$ cd ~/src/Firmware
$ make posix_sitl_default gazebo
```

O simulador só irá funcionar caso este tenha sido instalado. Para configurá-lo deve-se seguir o tutorial em https://dev.px4.io/en/simulation/gazebo.html. Uma alternativa caso não queira instalar o *Gazebo* é utilizar o *JMAVsim*, para isso basta substituir a palavra *gazebo* por *jmavsim* sempre que compilar o código para o simulador.

B.1.2 Implementando um controlador de teste: *test_controller*

Uma vez com o *firmware* instalado e configurado, acesse a pasta onde contêm os módulos da PX4 e crie uma nova pasta para o controlador chamada *test_controller*:

```
$ cd ~/src/Firmware/src/modules
$ mkdir test_controller
```

Em seguida deve ser criado o código fonte do controlador. Pode ser usado qualquer editor de texto mas uma IDE é altamente recomendada. Para simplificar será utilizado o *gedit* neste trabalho:

```
$ cd ~/src/Firmware/src/modules/test_controller
$ gedit "test_controller.cpp"
```

O código fonte do controlador utilizado neste tutorial está abaixo. Sinta-se livre para modificá-lo e usar uma abordagem diferente. Deve-se estar ciente que todos os tópicos estão na pasta */src/Firmware/msg*. Entender os tópicos é uma ferramenta extramemente poderosa pois possibilita a aquisição de diversos tipos de mensagens vindas de sensores e de outros módulos importantes (*EKF*, RC, etc).

O *test_controller* é um controlador PID com a mesma arquitetura do controlador da *PX4* (mc_att_control), sendo um controlador P para os ângulos e um controlador PD para velocidade angular, para veículos do tipo quadrirotor.

Acesse https://github.com/leoavelino/tcc/blob/master/test_controller.cpp copie o código fonte, cole e salve no arquivo *test_controller.cpp*. Agora deve-se criar um arquivo *CMakeLists.txt* na pasta do controlador que será responsável por compilar os arquivos deste novo módulo:

```
$ cd ~/src/Firmware/src/modules/test_controller
$ gedit "CMakeLists.txt"
```

Copie o conteúdo abaixo no arquivo e salve-o na pasta do controlador:

```
px4_add_module(
        MODULE modules__test_controller
        MAIN test_controller_main
        COMPILE_FLAGS
        STACK_MAIN 2000
        SRCS
                test_controller.cpp
        DEPENDS
                platforms__common
)
```

Antes de testar o controlador repare que o *test_controller* está publicando no tópico *actuators_control*, cujo os dados irão para o módulo do mixer. Porém este tópico também é usado pelo controle padrão da PX4 e desta maneira irá existir um conflito pois o mixer irá receber dados de dois controladores. Portanto, abra o código fonte do controlador principal:

```
$ gedit ~/src/Firmware/src/modules/mc_att_control/mc_att_control_main.cpp
```

Agora procure as seguintes linhas onde os comandos estão sendo publicados:

```
/* publish controller status */
if (_controller_status_pub != nullptr) {
        (ORB_ID(mc_att_ctrl_status), _controller_status_pub,
        &_controller_status);

} else {
        _controller_status_pub = orb_advertise(ORB_ID(mc_att_ctrl_status),
        &_controller_status);
}
```

Comente todas as linhas acima. Para isto basta adicionar os caracteres // na frente de cada uma das linhas. O resultado final então é:

```
/* publish controller status */
//if (_controller_status_pub != nullptr) {
//      (ORB_ID(mc_att_ctrl_status), _controller_status_pub,
//      &_controller_status);

//} else {
//      _controller_status_pub = orb_advertise(ORB_ID(mc_att_ctrl_status),
//      &_controller_status);
//}
```

Salve e feche o arquivo. Desta maneira os torques calculados pelo *mc_att_control* não serão publicados no tópico *actuators_control*, e portanto somente o *test_controller* irá utilizá-lo.

Uma outra alternativa seria desativar completamente o módulo *mc_att_control*. Porém, não é recomendado fazer isso antes de estudar cautelosamente o que este módulo faz, pois entre outras coisas, este módulo toma conta de mudar referências de entrada de determinados modos de vôo como *RATTITUDE* e *ACRO*.

Em seguida deve-se acessar deve-se avisar ao compilador que o módulo do novo controlador inserido deve ser compilado também. Para a *Pixhawk* deve-se digitar na linha de comando: (caso objetivo seja utilizar no simulador, pule esta parte).

```
$ gedit ~/src/Firmware/cmake/configs/nuttx_px4fmu-v2_default.cmake
```

E inserir uma linha com a o diretório onde está o novo módulo conforme abaixo:

```
        #
        # Vehicle Control
        #
        modules/fw_att_control
        modules/fw_pos_control_l1
        modules/gnd_att_control
        modules/gnd_pos_control
```

```
        modules/mc_att_control
        modules/mc_pos_control
        modules/vtol_att_control
        modules/test_controller
```

Agora, para iniciar o módulo toda vez em que o sistema for iniciado deve-se abrir o script de inicialização dos controladores. Digite no terminal:

```
$ gedit ~/src/Firmware/ROMFS/px4fmu_common/init.d/rc.mc_apps
```

Encontre a linha onde o *mc_att_control* está sendo inicializado e adicione o novo controlador:

```
        #---------------------------------------

        mc_att_control start
        test_controller start
        mc_pos_control start

        #
```

Caso o objetivo seja utilizar o simulador deve-se digitar:

```
$ gedit ~/src/Firmware/cmake/configs/posix_sitl_default.cmake
```

E em seguida adicionar o controlador conforme abaixo:

```
        #
        # Vehicle Control
        #
        modules/fw_att_control
        modules/fw_pos_control_l1
        modules/gnd_att_control
        modules/gnd_pos_control
        modules/mc_att_control
        modules/mc_pos_control
        modules/vtol_att_control
        modules/test_controller
```

Para iniciar o módulo na inicialização do sistema digite:

```
$ gedit ~/src/Firmware/posix-configs/SITL/init/ekf2/iris
```

E adicione a seguinte linha:

```
mc_att_control start
test_controller start
mixer load /dev/pwm_output0 ROMFS/px4fmu_common/mixers/quad_dc.main.mix
```

Pronto, agora pode-se testar o controlador na *Pixhawk* digitando no terminal:

```
$ cd ~/src/Firmware/
$ make px4fmu-v2_default
```

Ou para o simulador:

```
$ cd ~/src/Firmware/
$ make posix_sitl_default gazebo
```

Para testar o controlador no simulador basta digitar no console da *PX4*:

```
> commander takeoff
```

Caso queira checar o status do novo controlador, deve-se digitar no console da *PX4*:

```
> test_controller status
```

B.2 Implementando um novo *mixer* na *PX4*

Dependendo do tipo de aplicação, o controlador implementado acima pode não resultar em algo desejado pois o mesmo está utilizando o *mixer* da *PX4*. O *mixer* existente força os valores de torques (em *roll*, *pitch* e *yaw*) a estarem entre -1 e 1, e o valor de *throttle* é forçado entre valores de 0 e 1. Isto significa que se o *test_controller* calcular valores de torque entre 2 e 20 por exemplo, esses valores sempre estarão saturados em 1.

Para ter o controle total da aplicação, e publicar valores de *PWM* na saída, o código do mixer deve ser *hackeado*. Antes de começar esta aplicação, lembre-se de fazer um *backup*.

Tenha em mente que o *mixer* da *PX4* é bem eficiente, e que ele é feito de maneira que cada *frame* configurado aplique pesos diferentes para os angulos de atuação. Para ilustrar isto basta lembrar que os quadrirotores não são necessariamente simétricos.

O algoritmo do *mixer* mostrado neste tutorial não foi testado em campo. Foram feitos testes em *SITL*, e também, verificou-se com o auxílio do osciloscópio as saídas PWM da *Pixhawk* para validação deste tutorial, porém, este *mixer* nunca foi testado em vôo.

Primeiramente, acesse o código-fonte do *mixer*:

```
$ gedit ~/src/Firmware/src/modules/systemlib/mixer/mixer_multirotor.cpp
```

Procure a implementação da função *MultirotorMixer::mix(float ...)*. Repare que o *mixer* lê os valores torque através da função *get_control()*, calcula os valores de saída para cada motor (saídas entre -1 e 1) e copia para a variável *outputs*[]. Vá até a última linha do código e acima dela implemente o *mixer* simples conforme mostrado abaixo:

```
                //Notify saturation status
        if (status_reg != nullptr) {
```

```
                (*status_reg) = _saturation_status.value;
        }

        //ler valor de roll publicado no topico actuators_control
        float roll_t = get_control(0, 0);
        //ler valor de pitch publicado no topico actuators_control
        float pitch_t = get_control(0, 1);
        //ler valor de yaw publicado no topico actuators_control
        float yaw_t = get_control(0, 2);
        //ler valor de throttle publicado no topico actuators_control
        float thrust_t = get_control(0, 3);

        //motor frontal direito
        outputs[0] = -roll_t+pitch_t+yaw_t+thrust_t;
        //motor frontal esquerdo
        outputs[1] = roll_t-pitch_t+yaw_t+thrust_t;
        //motor traseiro esquerdo
        outputs[2] = roll_t+pitch_t-yaw_t+thrust_t;
        //motor traseiro direito
        outputs[3] = -roll_t-pitch_t-yaw_t+thrust_t;

        return _rotor_count;
}
```

Você pode comentar todo o resto do código desta função, ou simplesmente deixar, já que a variável *outputs[]* é sobre escrevida no final da função, e portanto, o cálculo realizado acima não irá fazer diferença. Vale lembrar que este *mixer* não é confiável pois está considerando um quadrirotor simétrico, e sua implementação é apenas didática.

O *mixer* padrão agora não é mais executado, e portanto os valores de torque publicados pelo controlador não serão mais delimitados. Porém, deseja-se alterar também como os valores "mixados"aos motores são convertidos em *PWM*. Existem duas abordagens, sendo que a primeira é para a *Pixhawk* e a segunda para o simulador.

B.2.1 *Hackeando* o driver *Pixhawk*

Para mudar a maneira em que os comandos para os motores são convertidos em *PWM*, deve-se acessar primeiramente o código fonte que delimita os valores de *PWM*. Digite no terminal

```
$ gedit ~/src/Firmware/src/modules/systemlib/pwm_limit/pwm_limit.c
```

Neste código-fonte acontece a mudança na máquina de estados da *PWM*. Não é o objetivo deste tutorial modificar a máquina de estados da *PX4*, o objetivo é apenas editar a maneira em que ela calcula o *PWM* para os motores caso a máquina de estados esteja

em modo *ON*. Para saber mais sobre a máquina de estados acesse https://dev.px4.io/en/concept/pwm_limit.html. Encontre a linha de código e edite conforme abaixo:

```
case PWM_LIMIT_STATE_ON:

for (unsigned i = 0; i < num_channels; i++) {

        float control_value = output[i];

        /* check for invalid / disabled channels */
        if (!isfinite(control_value)) {
                effective_pwm[i] = disarmed_pwm[i];
                continue;
        }

        if (reverse_mask & (1 << i)) {
                control_value = -1.0f * control_value;
        }

        effective_pwm[i] = control_value * (max_pwm[i] - min_pwm[i]) / 2
                + (max_pwm[i] + min_pwm[i]) / 2;

        /* last line of defense against invalid inputs */
        if (effective_pwm[i] < min_pwm[i]) {
                effective_pwm[i] = min_pwm[i];

        } else if (effective_pwm[i] > max_pwm[i]) {
                effective_pwm[i] = max_pwm[i];
        }

}

break;
```

Atenção que ao ignorar os limites de *PWM* passados por referência *max_pwm*[], *min_pwm*[], os parâmetros do sistema *PWM_MAX* e *PWM_MIN*, perdem o efeito.

B.2.2 *Hackeando* o driver do Simulador

Para mudar a maneira em que os comandos para os motores são convertidos em *PWM*, acesse o driver de saída *PWM* para o simulador no terminal:

```
$ gedit ~/src/Firmware/src/src/drivers/pwm_out_sim/pwm_out_sim.cpp
```

Encontre a linha de código onde os comandos são transformados em *PWM* e edite conforme abaixo (cor vermelha apenas):

```
/* iterate actuators */
```

```
for (unsigned i = 0; i < num_outputs; i++) {
        /* last resort: catch NaN, INF and out-of-band errors */
        if (i < outputs.noutputs &&
        PX4_ISFINITE(outputs.output[i]) &&
        outputs.output[i] >= -5.0f &&
        outputs.output[i] <= 5.0f) {
                /* scale for PWM output 1000 - 2000us */
                // saida PWM para o motor i
                outputs.output[i] = 1500 + (100 * outputs.output[i]);

        } else {
                /*
                * Value is NaN, INF or out of band - set to the minimum value.
                * This will be clearly visible on the servo status and will
                * limit the risk of accidentally spinning motors.
                * It would be deadly in flight.
                */
                outputs.output[i] = PWM_SIM_DISARMED_MAGIC;
        }
}

/*
* O codigo acima esta dizendo que para valores de motores publicados
* pelo mixer no range -5.0 a 5.0, o PWM gerado para os motores (de 0 a 3)
* sera de 1500 + (100 * outputs.output[i]), sendo que outputs.output[i] eh o
* valor de comando publicado pelo mixer para aquele motor
*
* Eh definido um range entre 1000 e 2000 de PWM
*/
```

Caso depois de alterar o código do mixer e do driver o sistema exibe no console mensagens de *Freefall detected*, pode-se corrigir o problema mudando um parâmetro de velocidade mínima para veículo em terra. Digite no terminal:

```
$ gedit ~/src/Firmware/src/src/modules/land_detector/land_detector_params.cpp
```

Procure o seguinte parâmetro e altere seu valor para 0, conforme abaixo:

```
PARAM_DEFINE_FLOAT(LNDMC_Z_VEL_MAX, 0.00f);
```